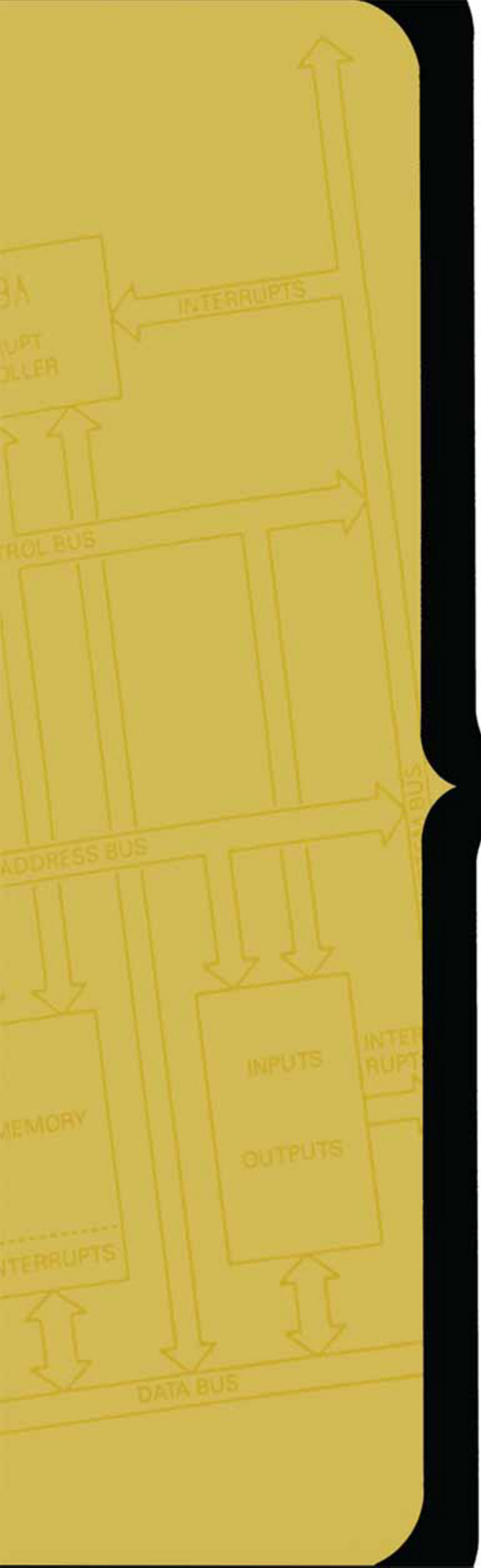

INTERRUPTS
ADDRESS BUS
INPUTS
OUTPUTS
MEMORY
INTERRUPTS
DATA BUS

8086–8088 Architecture and Programming

Including 8087 numeric processor

J-M Trio

8086–8088 Architecture and Programming
Including 8087 numeric processor

Macmillan Computer Science Series

Consulting Editor
Professor F.H. Sumner, University of Manchester

S.T. Allworth, *Introduction to Real-time Software Design*
Ian O. Angell, *A Practical Introduction to Computer Graphics*
R.E. Berry and B.A.E. Meekings, *A Book on C*
G.M. Birtwistle, *Discrete Event Modelling on Simula*
T.B. Boffey, *Graph Theory in Operations Research*
Richard Bornat, *Understanding and Writing Compilers*
J.K. Buckle, *Software Configuration Management*
W.D. Burnham and A.R. Hall, *Prolog Programming and Applications*
J.C. Cluley, *Interfacing to Microprocessors*
Robert Cole, *Computer Communications*
Derek Coleman, *A Structured Programming Approach to Data*
Andrew J.T. Colin, *Fundamentals of Computer Science*
Andrew J.T. Colin, *Programming and Problem-solving in Algol 68*
S.M. Deen, *Fundamentals of Data Base Systems*
S.M.Deen, *Principles and Practice of Database Systems*
P.M. Dew and K.R. James, *Introduction to Numerical Computation in Pascal*
M.R.M. Dunsmuir and G.J. Davies, *Programming the UNIX System*
K.C.E. Gee, *Introduction to Local Area Computer Networks*
J.B. Gosling, *Design of Arithmetic Units for Digital Computers*
Roger Hutty, *Fortran for Students*
Roger Hutty, *Z80 Assembly Language Programming for Students*
Roland N. Ibbett, *The Architecture of High Performance Computers*
Patrick Jaulent, *The 68000 — Hardware and Software*
J.M. King and J.P. Pardoe, *Program Design Using JSP — A Practical Introduction*
H. Kopetz, *Software Reliability*
E.V. Krishnamurthy, *Introductory Theory of Computer Science*
V.P. Lane, *Security of Computer Based Information Systems*
Graham Lee, *From Hardware to Software — an introduction to computers*
A.M. Lister, *Fundamentals of Operating Systems, third edition*
G.P. McKeown and V.J. Rayward-Smith, *Mathematics for Computing*
Brian Meek, *Fortran, PL/1 and the Algols*
Derrick Morris, *System Programming Based on the PDP11*
P. Oets, *Using MS-DOS and PC-DOS*
Christian Queinnec, *LISP*
John Race, *Case Studies in Systems Analysis*
W.P. Salman, O. Tisserand and B. Toulout, *FORTH*
L.E. Scales, *Introduction to Non-linear Optimization*
Peter S. Sell, *Expert Systems — A Practical Introduction*
Colin J. Theaker and Graham R. Brookes, *A Practical Course on Operating Systems*
M.J. Usher, *Information Theory for Information Technologists*
B.S. Walker, *Understanding Microprocessors*
Peter J.L. Wallis, *Portable Programming*
I.R. Wilson and A.M. Addyman, *A Practical Introduction to Pascal — with BS6192, second edition*

8086–8088 Architecture and Programming

Including 8087 numeric processor

Jean-Michel Trio
Centre National de la Recherche Scientifique
Strasbourg

M
MACMILLAN

© Editions EYROLLES, Paris 1984

Authorised English Language edition
(with additions and revisions) of
Microprocesseurs 8086/8088 —
Architecture et Programmation —
Coprocesseur de calcul 8087
by Jean-Michel Trio, first published 1984 by
Editions EYROLLES, 61 boulevard Saint-Germain,
75005 Paris

Translated by M.J. Stewart

© English Language edition, Macmillan Publishers Ltd, 1985

All rights reserved. No reproduction, copy or transmission of this publication may be made without written permission.

No paragraph of this publication may be reproduced, copied or transmitted save with written permission or in accordance with the provisions of the Copyright Act 1956 (as amended).

Any person who does any unauthorised act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

First published 1985

Published by
Higher and Further Education Division
MACMILLAN EDUCATION LTD
Houndmills, Basingstoke, Hampshire RG21 2XS
and London
Companies and representatives
throughout the world

British Library Cataloguing in Publication Data
Trio, Jean-Michel
8086–8088 architecture and programming : including
8087 numeric co-processor.—(Macmillan computer
studies)
1. INTEL 8086 (Microprocessor) 2. INTEL 8088
(Microprocessor)
I. Title II. Microprocesseurs 8086–8088–
architecture et programmation. *English*
001.64'04 QA76.8.1292

ISBN 978-0-333-39692-6 ISBN 978-1-349-08186-8 (eBook)
DOI 10.1007/978-1-349-08186-8

Contents

Preface ix

1 General Introduction 1

1.1 Hardware Organisation 1
- 1.1.1 Minimum system 1
- 1.1.2 Interrupt handling 4
- 1.1.3 System with numeric co-processor and/or I/O 5
- 1.1.4 Multi-user bus system 7

1.2 Software Organisation 7
- 1.2.1 Minimum organisation 7
- 1.2.2 Structured organisation 9
- 1.2.3 Multiple segments 10
- 1.2.4 Multiple modules 11
- 1.2.5 Information grouping 12

2 Program Structures 16

2.1 Segment 16
- 2.1.1 Creation of the segment 16
- 2.1.2 Generation of the offset in the segment 18
- 2.1.3 Loading CPU segment registers and pointers 23

2.2 The PROC/ENDP Directive 29

2.3 GROUP Directive 30

2.4 The Link Between Modules 33
- 2.4.1 NAME directive 33
- 2.4.2 PUBLIC directive 34
- 2.4.3 EXTRN directive 34
- 2.4.4 END directive 35

3 Definition and Initialisation of Data 36

3.1 Writing Identifiers 36

3.2 Identifier Attributes 36

3.3 Constants 37
- 3.3.1 Use in direct form in an instruction 37
- 3.3.2 EQU directive 37

3.4 Definition of Variables 38
- 3.4.1 Initialisation with a constant expression 38
- 3.4.2 Variable definition without initialisation 41
- 3.4.3 Pointer or index definition 41
- 3.4.4 Definition and initialisation of a data list 44
- 3.4.5 Values repeated several times 44

3.5 RECORD 45
- 3.5.1 Definition 45
- 3.5.2 Using RECORD for the definition and initialisation of variables 45
- 3.5.3 Use for the calculation of a constant expression 46
- 3.5.4 Operators associated with RECORD 47

3.6 Structure 48
- 3.6.1 Definition and use in reserving memory 48
- 3.6.2 Use with initialisation and initialisation overwriting 49
- 3.6.3 Reference to structured variables 51

4 Access to Data 53

4.1 Operands 53
- 4.1.1 Register type operands 54
- 4.1.2 Immediate operands 55
- 4.1.3 Memory type operands 56

4.2 Attribute Operators 62
- 4.2.1 Attribute corrector operators 62
- 4.2.2 Attribute use operators 64
- 4.2.3 Operators specific to RECORD 66
- 4.2.4 Arithmetic operators 67
- 4.2.5 Boolean operators 67
- 4.2.6 Binary operators 68
- 4.2.7 Operator hierarchy 68
- 4.2.8 EQU directive 68

4.3 The ASSUME Directive 70
- 4.3.1 Access in the same module 71
- 4.3.2 Access within a group 73
- 4.3.3 Intermodular access 75

4.4 Use of the Stack 77
- 4.4.1 Local variables 78
- 4.4.2 Parameter passing by the stack 79
- 4.4.3 Pascal-assembler link 81
- 4.4.4 FORTRAN-assembler link 86
- 4.4.5 BASIC 86-assembler link 91

5 Circuit Description 101

5.1 Description of CPU 101
  5.1.1 Pin assignment 102
  5.1.2 Internal organisation 109

5.2 Bus Transfers 113
  5.2.1 Read and write cycle 113
  5.2.2 Organisation of memory space for the 8086 116

5.3 Circuits in the 8086/8088 family 119
  5.3.1 8284A clock generator 119
  5.3.2 8282/8283 8-bit address latch 121
  5.3.3 8286/8287 8-bit data transceivers 122
  5.3.4 8288 bus controller 123

5.4 Interrupts 125
  5.4.1 External interrupts 125
  5.4.2 Internal interrupts 127
  5.4.3 Interrupt pointer table 127
  5.4.4 Branch to interrupt routine 129
  5.4.5 8259A PIC - Programmable interrupt controller 130

6 Instruction Set 143

6.1 Data Transfer 144
  6.1.1 Explicit transfer 145
  6.1.2 Implicit transfer with the accumulator 145
  6.1.3 Address transfers 146
  6.1.4 Status word transfer 146

6.2 Arithmetic Operations 147
  6.2.1 Addition 147
  6.2.2 Subtraction 147
  6.2.3 Multiplication 149
  6.2.4 Division 149

6.3 Logic Operations 150

6.4 Repetitive Instructions (Strings) 152
  6.4.1 Operating mode 152
  6.4.2 Base instructions 152

6.5 Jump Instructions 154
  6.5.1 Unconditional transfers 155
  6.5.2 Iteration control 156
  6.5.3 Internal interrupts 156
  6.5.4 Conditional jumps 157

6.6 Processor Control 158
- 6.6.1 Operations on flags 158
- 6.6.2 Processor halt 159
- 6.6.3 Processor wait 159
- 6.6.4 Escape 159
- 6.6.5 Bus inhibit 159
- 6.6.6 Single step 160

7 8087 Numeric Processor Extension 163

7.1 Functional Description 163
- 7.1.1 System configuration 164
- 7.1.2 Sequence of bus operations 164
- 7.1.3 Representation of numbers 165

7.2 Microprocessor Architecture 169
- 7.2.1 Control unit 169
- 7.2.2 Numeric Execution Unit 170
- 7.2.3 Registers 171
- 7.2.4 Description and use of the stack 171
- 7.2.5 Status word 172
- 7.2.6 Tag words 175
- 7.2.7 Control word 175

7.3 Instruction Set 176
- 7.3.1 Data transfer instructions 176
- 7.3.2 Arithmetic instructions 178
- 7.3.3 Comparison instructions 183
- 7.3.4 Transcendental instructions 184
- 7.3.5 Constants 186
- 7.3.6 Processor control instructions 187

Appendix 192

Index 197

Preface

This book is intended to appeal to those engineers and technicians who wish to gain more detailed knowledge of the Intel 8086 and 8088 16-bit microprocessors, together with the associated 8087 numeric processor extension, which are currently installed in the majority of microcomputers and in many card systems.

The motivation for acquiring such knowledge may on the one hand be biased towards software in the case of system users keen to improve performance without ´lifting the bonnet´; on the other hand, it may be hardware oriented in the case of designers of intelligent machines or those entrusted with their maintenance. The latter category are often more interested in the development of complex architecture than in algorithm optimisation.

However, this simplistic categoristion is misleading and should be set aside. What software designer has not had to concern himself with the programming of a peripheral circuit controlling a dialogue protocol between his machine and its environment, or with a circuit controlling his system interrupts?

Where is the hardware designer who does not need to interface a high level language controlling a console with his assembler modules in order to make full use of the hardware performance capability of his system?

Of course, the ideal is to achieve a balance between the two approaches and to know how to make the appropriate choice between a hardware and a software solution for a given project.

This is the approach followed in this book, the content of which has been tested on a number of courses that have brought to light the need for many practical examples. Such examples are included from chapter 2 onwards, together with detailed commentaries.

All the programs included can be processed by the assembler in the form in which they are presented, and from example 4.15 onwards they can be executed.

It should be noted that, in contrast to the case with 8-bit machine language microprocessors, data directives to the assembler and executable instructions are by far the most important. One instruction can therefore be translated in many ways into machine code, depending on the context.

Indeed, in the case of short programs, it is by no means rare to write as many, if not more, directives as instructions.

Any reader tackling this topic for the first time is therefore urged to read the contents in the order presented and to study the examples with care, preferably trying them out on a machine. The author would be grateful to receive any comments or suggestions for improvements.

Both the author and the publishers would like to record their grateful appreciation and thanks to Aidan Loyns and Andy Vincent for their help in the preparation of the English language edition.

1 General Introduction

1.1 Hardware Organisation

1.1.1 Minimum system

The 8086 and 8088 microprocessors are both 16-bit devices that differ essentially only in the size of their external data buses - 16 bits for the 8086, 8 bits for the 8088. Software written for one processor will run on the other without modification. The advantage of producing a 16-bit processor that employs an external 8-bit data bus is that small system design is simplified. The price that must be paid for this simplification is that, since program and data may only be fetched from memory 8 bits at a time, a program will take longer to execute on the 8088. With the exception of some hardware aspects, the basic similarity between the 8086 and 8088 allows the two processors to be considered together. This chapter will seek to give a general overview of both the hardware and software aspects of the 8086/8088 processors.

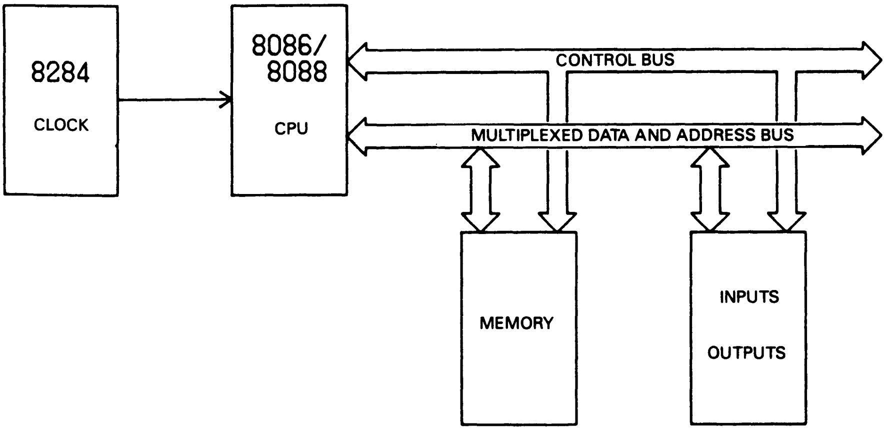


Figure 1.1

J.-M. Trio, *8086–8088 Architecture and Programming*
© English Language edition, Macmillan Publishers Ltd 1985

Data registers

(accessible as 16-bit or 8-bit registers)

	15 8	7 0	
AX	AH	AL	Accumulator
BX	BH	BL	Base
CX	CH	CL	Count
DX	DH	DL	Data

Pointer and index registers

15 0	
BP	Base pointer
SI	Source index
DI	Destination index
SP	Stack pointer
IP	Instruction pointer

Segment registers

15 0	
CS	Code segment
DS	Data segment
SS	Stack segment
ES	Extra segment

Status word

15 8	7 0
H flags	L flags

It is useful to start by outlining the register structure of the processors. Although the registers will be considered in some detail in chapter 6, it is helpful to summarise them at this stage in order to clarify the discussion that follows.

The registers accessible to the programmer are grouped in four categories.

The CPU is synchronised both internally and to the rest of the system by a crystal oscillator. In order to simplify the design of an appropriate circuit a special chip, the 8284, is provided as part of the 8086/8088 chip set. This device will be considered in more detail in chapter 5.

In common with most microprocessors, the 8086/8088 employs the three buses described below.

(1) The control bus mainly comprises the signals to control reading, writing, and distinguishing between memory and input/output (I/O) references.

(2) The address bus is made up of 20 lines and thus permits access to 1 Megabyte (1,048,576 bytes) of memory. Since it is unnecessary to be able to access such an extensive range of I/O locations, I/O references are limited to 16 bits.

Internally, both the 8086 and 8088 microprocessors employ 16-bit buses for data and addresses. In the 8088 the internal data bus is converted to the external 8-bit data bus by a bus interface unit (BIU). The BIU contains all the differences between the 8086 and 8088 and will be returned to in a later chapter.

Despite the presence of a 20-bit external address bus, both processors employ only an internal 16-bit address bus in order to retain compatibility with the processors´ 16-bit registers. The 20-bit external

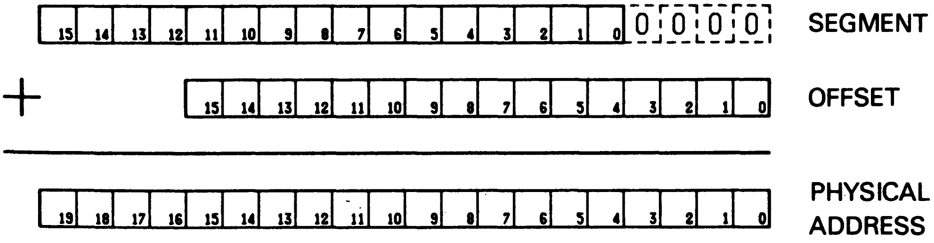


Figure 1.2

address is formed from two 16-bit quantities, a segment address and an offset. The segment portion of the address is obtained from one of the four segment registers, and is multiplied by 16 (shifted four places to the left) before being added to the offset.

The combination rule is as shown in figure 1.2.

The resulting 20-bit address allows the full memory space to be accessed. Producing an address in this way means that the same physical address may be formed with a great many combinations of segments and offsets. However, the chief advantage of this method of address formation is that all programs may be written to start from address zero, added to which several such programs may be loaded into memory at the same time.

To differentiate between programs, a separate segment address is assigned to each program. The segment address is that required to allow the physical start of the program code to be accessed with an initial offset of zero. While the code segment register is always used for program code references, the other three segment registers, which do have default functions, may be used more flexibly.

The CPU therefore retains permanently the following four segments: CS for the code, DS and ES for the data and variables and SS for the stack. It can therefore access 4 x 64K bytes at any time. If physical addresses outside this range are to be accessed, it must be done by modifying one of the segment registers.

(3) The data bus, consisting of 8 or 16 lines. This bus is multiplexed with the address bus. The schematic system shown in figure 1.1 would require the memory and I/O devices to demultiplex the address and data if it were to operate. A system where the address and data are demultiplexed is shown in figure 1.3. In this circuit 8282/8283 latches are used to retain the active address, and one or two 8286/8287 bidirection bus transceivers are used to control the data bus - two transceivers being required for the 16-bit data bus of the 8086.

1.1.2 Interrupt handling

In order to synchronise the execution of a program with external events, the CPU is provided with interrupt inputs. The 8086/8088 have one maskable, general-purpose input and one non-maskable, high priority input. The latter is reserved for the detection of exceptional events, such as power supply failure and other emergencies.

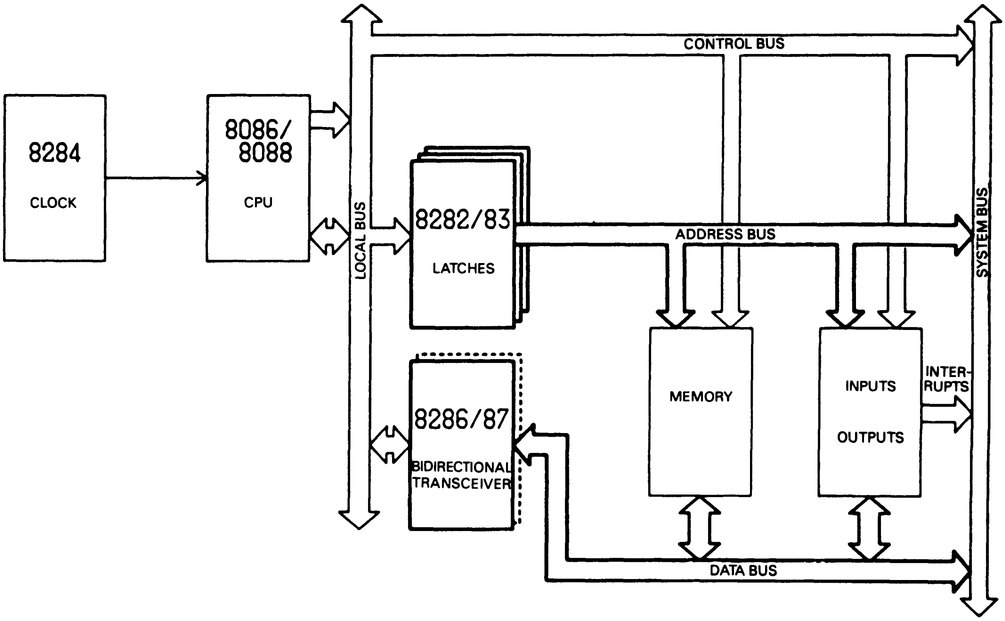

8284
CLOCK
8086/
8088
CPU
LOCAL BUS
CONTROL BUS
8282/83
LATCHES
ADDRESS BUS
SYSTEM BUS
8286/87
BIDIRECTIONAL
TRANSCEIVER
MEMORY
INPUTS
OUTPUTS
INTER-
RUPTS
DATA BUS

Figure 1.3

The handling of maskable interrupts is normally delegated to a special circuit that can handle up to eight interrupt inputs. Each of these inputs can itself receive the output of another identical controller, thus allowing up to 64 interrupts.

The purpose of the controller is to latch pending interrupts and select the highest priority input pending according to the priority and masking determined by the program.
It then forwards the interrupt request to the CPU and records the acknowledgement of it.

During execution of this acknowledgement, it gives to the CPU, via the data bus, a number corresponding to the interrupt line handled.
The CPU translates this number into an address in the first 1K page of memory and fetches from it the segment word and the offset word of the corresponding interrupt service routine subroutine.

1.1.3 System with numeric co-processor and/or I/O

Each co-processor is capable of taking control of the system instead of the CPU; it is therefore connected to a local multimaster bus.

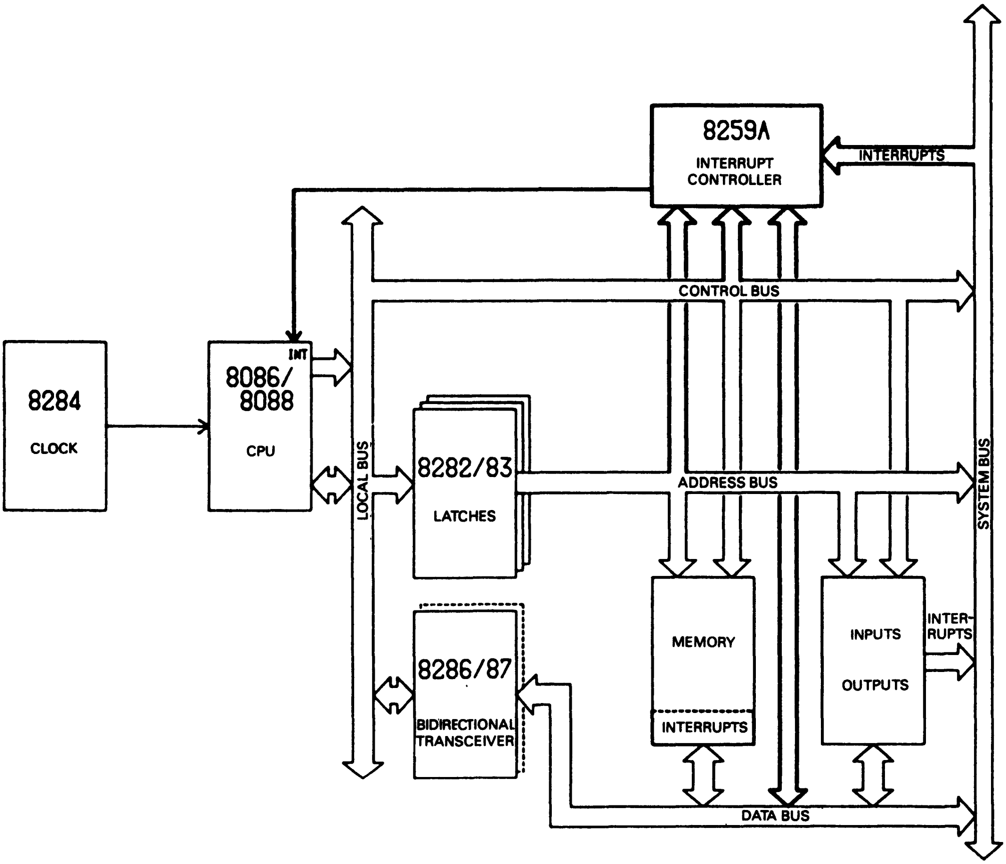

8259A
INTERRUPT CONTROLLER
INTERRUPTS
CONTROL BUS
8284
CLOCK
INT
8086/8088
CPU
LOCAL BUS
8282/83
LATCHES
ADDRESS BUS
SYSTEM BUS
8286/87
BIDIRECTIONAL TRANSCEIVER
MEMORY
INTERRUPTS
INPUTS
OUTPUTS
INTER-RUPTS
DATA BUS

Figure 1.4 Interrupt handling

The handling of control signals is delegated to a special circuit, the 8288 bus controller, in order to release a certain number of pins on the CPU and provide sufficient power on the control bus.

The freed pins allow co-processors to follow the CPU´s action permanently and request system control from it.

The combination of numeric co-processor and 8086/8088 is only visible to the programmer from the extension to the CPU instruction set, if working in assembler, or by a library change if working in a high-level language. All that is required is for some details of synchronisation and interrupt in the event of calculation error to be taken into account in the event of writing programs in assembler.

The 8089 I/O co-processor is itself an autonomous processor that can have its own independent memory and

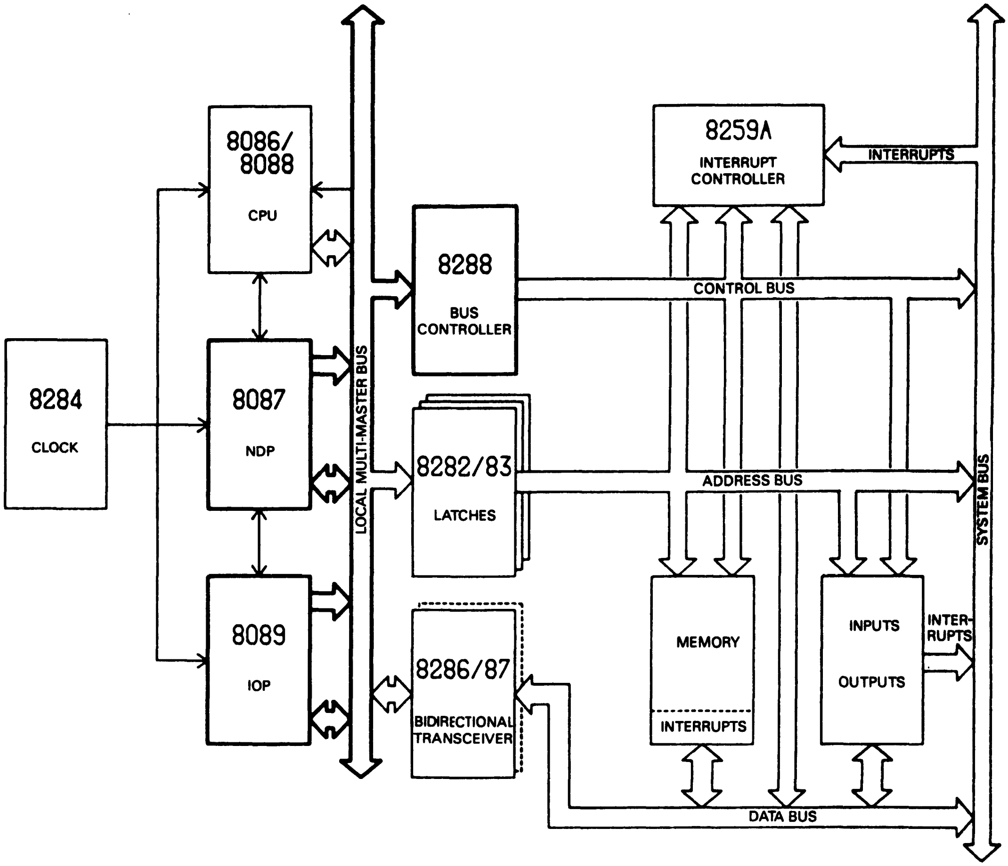

8086/
8088
CPU
8284
CLOCK
8087
NDP
8089
IOP
LOCAL MULTI-MASTER BUS
8288
BUS CONTROLLER
8282/83
LATCHES
8286/87
BIDIRECTIONAL TRANSCEIVER
8259A
INTERRUPT CONTROLLER
INTERRUPTS
CONTROL BUS
ADDRESS BUS
DATA BUS
MEMORY
INTERRUPTS
INPUTS
OUTPUTS
INTER-
RUPTS
SYSTEM BUS

Figure 1.5 System with numeric co-processor and/or I/O

I/O system. All that is required is to have a memory area in common with the CPU for the exchange of housekeeping information.

1.1.4 Multi-user bus system

When the CPU and its co-processors are grouped on a bus system whose use can be requested by another CPU or by one or more 8089 devices functioning in remote mode, a bus arbitration unit, the 8289, is required in order to avoid bus conflicts (see figure 1.6).

1.2 Software Organisation

1.2.1 Minimum organisation

This is illustrated in figure 1.7.

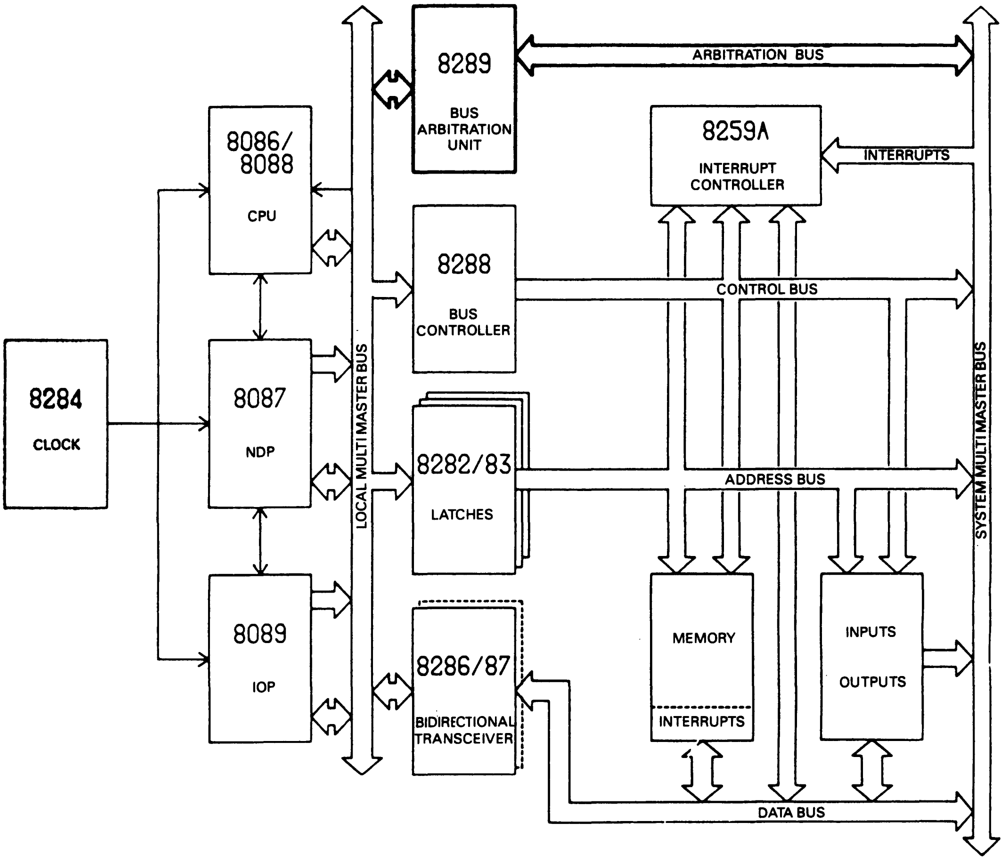

8289
BUS ARBITRATION UNIT
ARBITRATION BUS
8086/8088
CPU
8259A
INTERRUPT CONTROLLER
INTERRUPTS
8288
BUS CONTROLLER
CONTROL BUS
8284
CLOCK
8087
NDP
LOCAL MULTI MASTER BUS
8282/83
LATCHES
ADDRESS BUS
SYSTEM MULTI MASTER BUS
8089
IOP
8286/87
BIDIRECTIONAL TRANSCEIVER
MEMORY
INTERRUPTS
INPUTS
OUTPUTS
DATA BUS

Figure 1.6

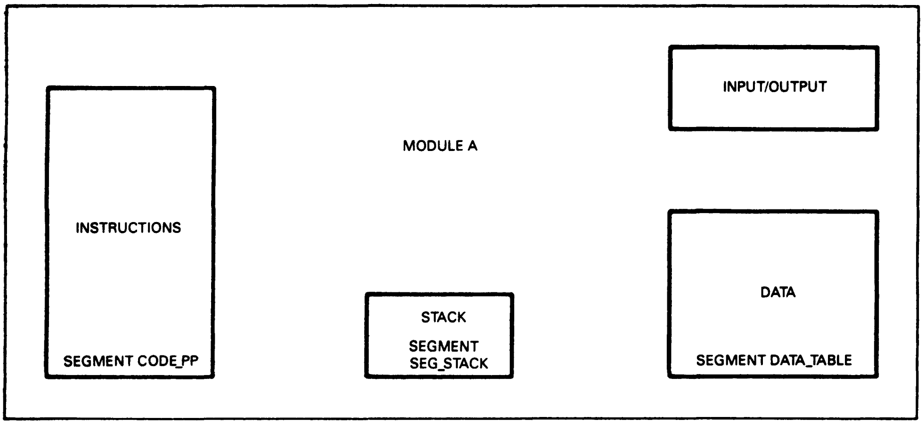

MODULE A
INSTRUCTIONS
SEGMENT CODE_PP
STACK
SEGMENT
SEG_STACK
INPUT/OUTPUT
DATA
SEGMENT DATA_TABLE

Figure 1.7

The instructions are grouped in a segment whose base address is held by the code segment (CS) register of the CPU.

The program variables and constants are grouped in another segment that is referenced by one of two data segment registers of the CPU.

The I/O circuits are the peripheral circuits of the CPU and are positioned at addresses fixed by the wiring and not by the assembler. There are rarely very many of them and certainly their numbers never exceed 64K; the CPU can therefore handle them ´outside segment´ by only activating 16 of the 20 lines of its address bus. However, this specialised I/O system is very limited; there are only two instructions in the instructions set that are applicable to it. For this reason input/outputs are often regarded as ´memories´ and thus form a segment of ordinary ´variables´. It will then be sufficient to cause the assembler to look after the allocation of addresses.

Finally, a stack is required for temporary storage during execution of a program. It lies in a distinct segment whose base address is held by the stack segment (SS) of the CPU.

All this information is grouped into a MODULE which is held in a file by the operating system. It is this file that the assembler operates on. All that is required is to pass the name of this file to the assembler.

1.2.2 Structured organisation

The programmer soon finds it necessary to structure his program and distinguish a main program from subroutines, called procedures. These enable a task that is often repeated to be written only once but which can then be called on from any part of the program.

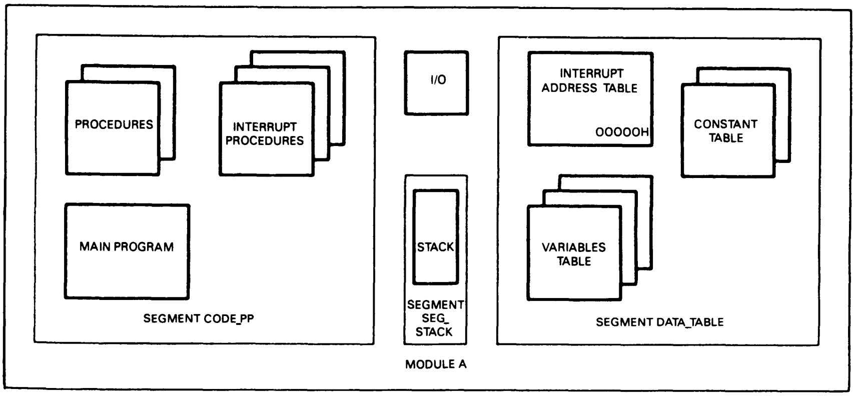

PROCEDURES
INTERRUPT PROCEDURES
MAIN PROGRAM
SEGMENT CODE_PP
I/O
STACK
SEGMENT SEG_ STACK
INTERRUPT ADDRESS TABLE
00000H
CONSTANT TABLE
VARIABLES TABLE
SEGMENT DATA_TABLE
MODULE A

Figure 1.8

In this organisation, the stack becomes very important because it will control the passing of the parameters of one program to another; it will also keep the data that is indispensable for transferring control of one task to another - the return address, CPU status, data values, and so on.

The variables can also be structured logically in different arrays. For example, constants do not have the same logical function as variables which are liable to be modified at any time by the program.

Similarly, the table of interrupt addresses, which responds to action of the interrupt controller, does not have to be mixed with other tables. (In fact, this table is necessarily located in the first kilobyte of memory (00000H), whereas the others can be located anywhere.)

1.2.3 Multiple segments

The last point made in the previous paragraph illustrates the need to extend beyond the framework of a single segment for variables and beyond another for

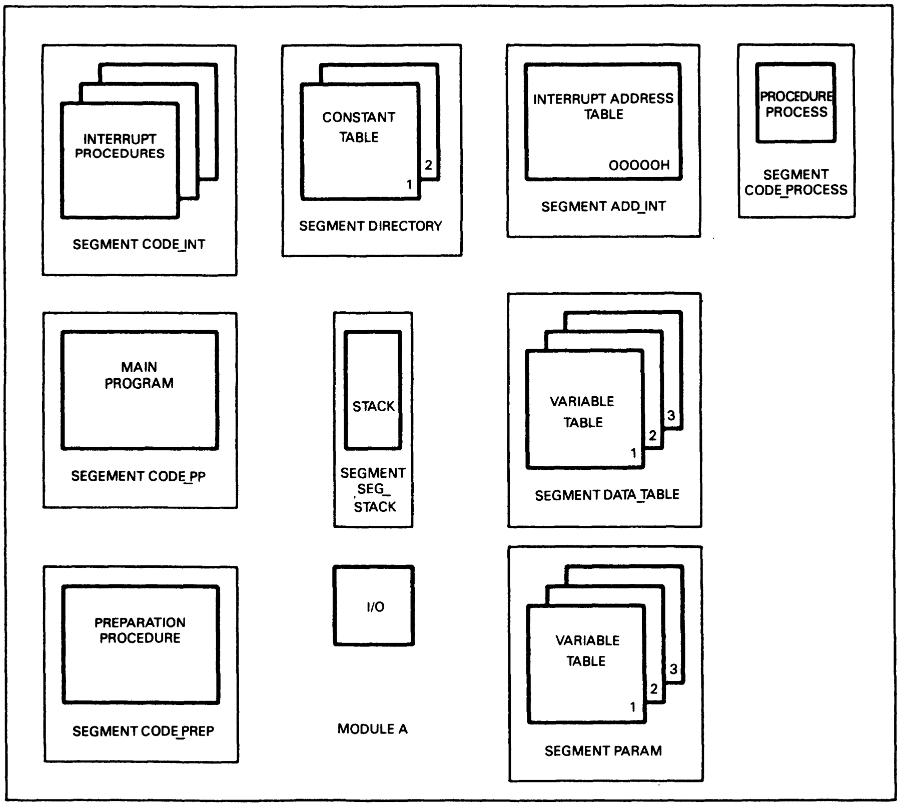

INTERRUPT PROCEDURES
SEGMENT CODE_INT
CONSTANT TABLE
2
1
SEGMENT DIRECTORY
INTERRUPT ADDRESS TABLE
OOOOOH
SEGMENT ADD_INT
PROCEDURE PROCESS
SEGMENT CODE_PROCESS
MAIN PROGRAM
SEGEMENT CODE_PP
STACK
SEGMENT SEG_ STACK
VARIABLE TABLE
3
2
1
SEGMENT DATA_TABLE
PREPARATION PROCEDURE
SEGMENT CODE_PREP
I/O
MODULE A
VARIABLE TABLE
3
2
1
SEGMENT PARAM

Figure 1.9

instructions. The size of the tables or programs may exceed 64K. They can therefore no longer be located in a single segment.

Note that, just as it was held to be convenient to group homogenous data into tables with distinct names, it can also seem preferable to group tables of the same type in different segments.

Figure 1.9 illustrates this new organisation.

1.2.4 Multiple modules

The next step is to extend the ideas to beyond the framework of a single module. The module is in fact placed in a unique file held by the operating system. When listing, assembling or compiling, all the information therefore has to be manipulated.

Why is it necessary to reassemble many times procedures that have no errors and have already been tested, when debugging subsequent procedures?

It is in fact more practical to divide the different pieces of the program into several modules as shown in figure 1.10.

In moving from one module to another it is possible to reopen a segment that has already been created by another module, or to create new segments in the module.

Only the stack segment generally remains unique for all modules. However, when size constraints are involved, it is still possible to define several stack segments.

This is particularly the case where recursive procedures are involved as they can be called a large number of times, thereby multiplying the size allocated to the passing of parameters. Finally, multi-module writing of a program is the norm when a mixture of languages is involved, such as BASIC, Pascal, FORTRAN, Assembler.

The object files generated by the assembler or compilers are then linked together by the link utility program which establishes the necessary connections for the intermodular transfer of data.

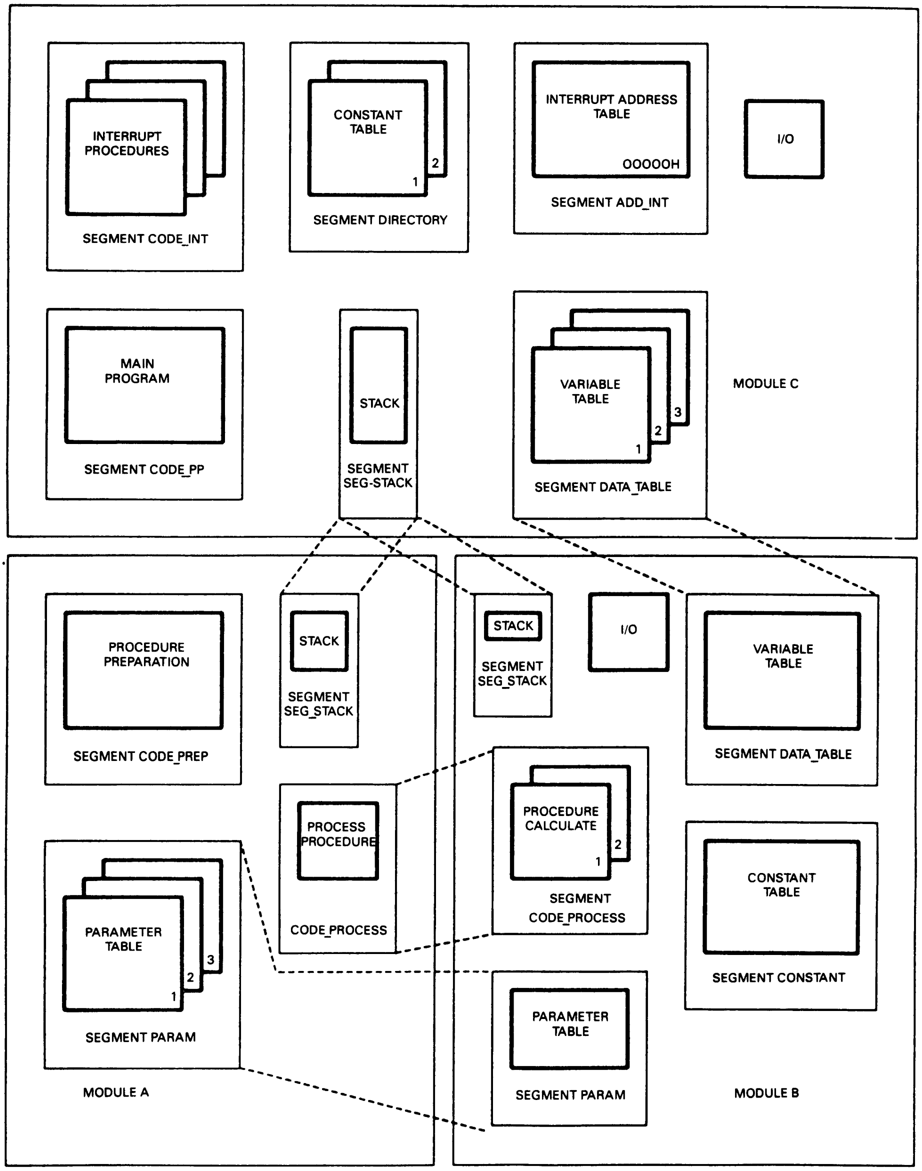

INTERRUPT PROCEDURES
SEGMENT CODE_INT
CONSTANT TABLE
2
1
SEGMENT DIRECTORY
INTERRUPT ADDRESS TABLE
00000H
SEGMENT ADD_INT
I/O
MAIN PROGRAM
SEGMENT CODE_PP
STACK
SEGMENT SEG-STACK
VARIABLE TABLE
3
2
1
SEGMENT DATA_TABLE
MODULE C
PROCEDURE PREPARATION
SEGMENT CODE_PREP
STACK
SEGMENT SEG_STACK
STACK
SEGMENT SEG_STACK
I/O
VARIABLE TABLE
SEGMENT DATA_TABLE
PROCESS PROCEDURE
CODE_PROCESS
PROCEDURE CALCULATE
2
1
SEGMENT CODE_PROCESS
PARAMETER TABLE
3
2
1
SEGMENT PARAM
CONSTANT TABLE
SEGMENT CONSTANT
PARAMETER TABLE
SEGMENT PARAM
MODULE A
MODULE B

Figure 1.10

1.2.5 Information grouping

It is clear however at this stage that the scattering of data can be inconvenient for programmers. There is no point in carefully arranging everything in different drawers and cupboards if it becomes tedious to control the different labels. The assembler has two solutions to this problem. First is a software solution - the group; the second a hardware one - the class.

Group

Each segment has its own segment address and, in order to pass from one segment to another, it is necessary for the CPU to alter one of its registers. Although this takes little time, this operation can become a handicap when the sum of the lengths of several segments does not exceed 64K.

In this case, the assembler handles them as a ´group´, working with a single ´segment´ value for all the segments. It sees to the addition of what the variable offsets of the different segments require in order for the absolute memory locations to be correctly attained (see figure 1.11).

Class

Apart from any software considerations, it can be useful, if not indispensable, to gather the elements of information into homogenous groups from the hardware point of view.

Thus, although program constants normally belong to a group of ´data´, they can be placed in read only memory with the instructions, and not in working storage with the variables.

Similarly, the set of code segments, whether grouped or not, is placed in an often enforced physical location.

Finally, the stack segment is generally placed at the highest addresses of working storage, since it is loaded dynamically by address decrementation. On the other hand, the variables are placed at low addresses in working storage so that they cover the interrupt table.

These considerations lead to the definition of a small number of classes used subsequently by the program loader or locator whose role is to assign all the physical addresses (see figure 1.12).

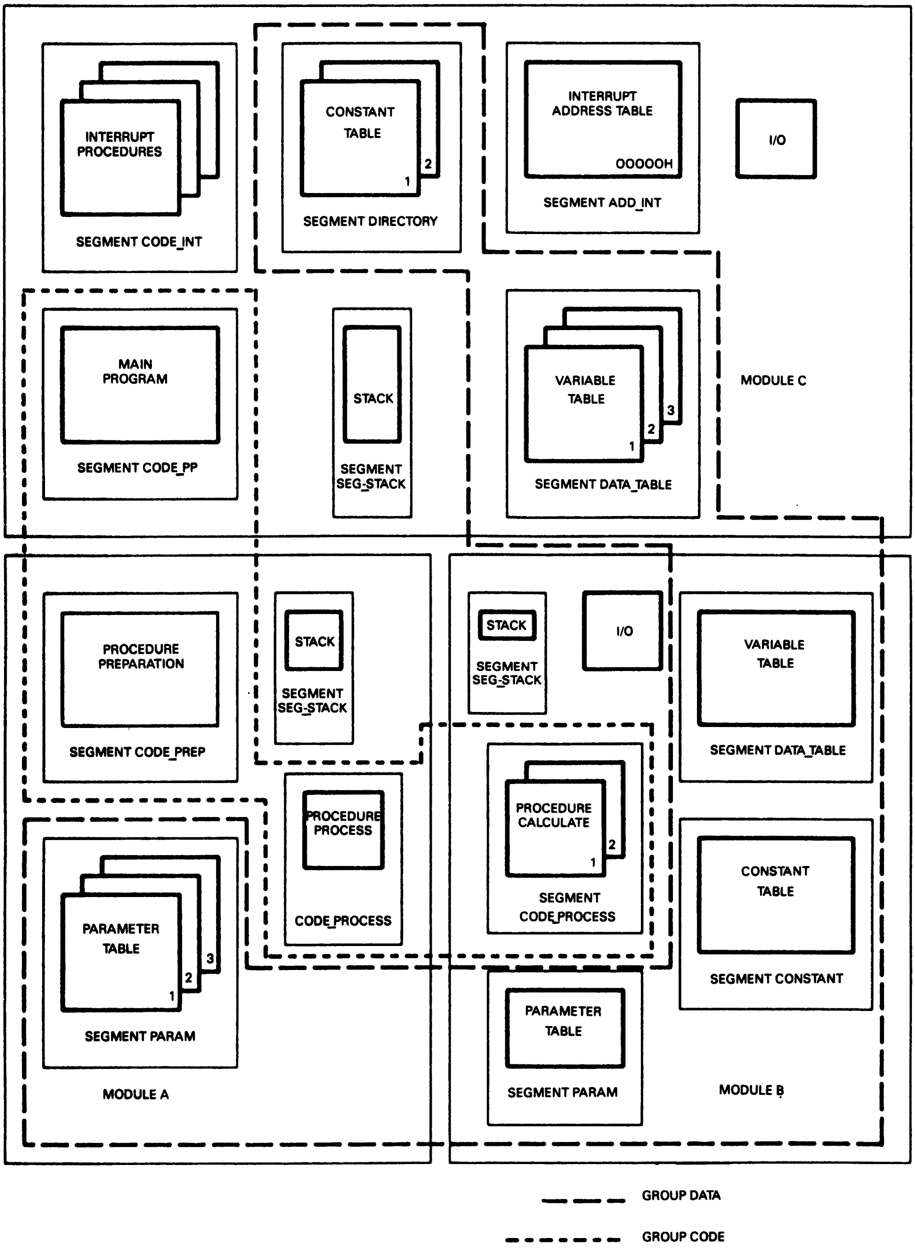

INTERRUPT PROCEDURES
SEGMENT CODE_INT
CONSTANT TABLE
1
2
SEGMENT DIRECTORY
INTERRUPT ADDRESS TABLE
00000H
SEGMENT ADD_INT
I/O
MAIN PROGRAM
SEGMENT CODE_PP
STACK
SEGMENT SEG_STACK
VARIABLE TABLE
1
2
3
SEGMENT DATA_TABLE
MODULE C
PROCEDURE PREPARATION
SEGMENT CODE_PREP
STACK
SEGMENT SEG_STACK
PROCEDURE PROCESS
CODE_PROCESS
PARAMETER TABLE
1
2
3
SEGMENT PARAM
MODULE A
STACK
SEGMENT SEG_STACK
I/O
PROCEDURE CALCULATE
1
2
SEGMENT CODE_PROCESS
VARIABLE TABLE
SEGMENT DATA_TABLE
CONSTANT TABLE
SEGMENT CONSTANT
PARAMETER TABLE
SEGMENT PARAM
MODULE B
GROUP DATA
GROUP CODE

Figure 1.11

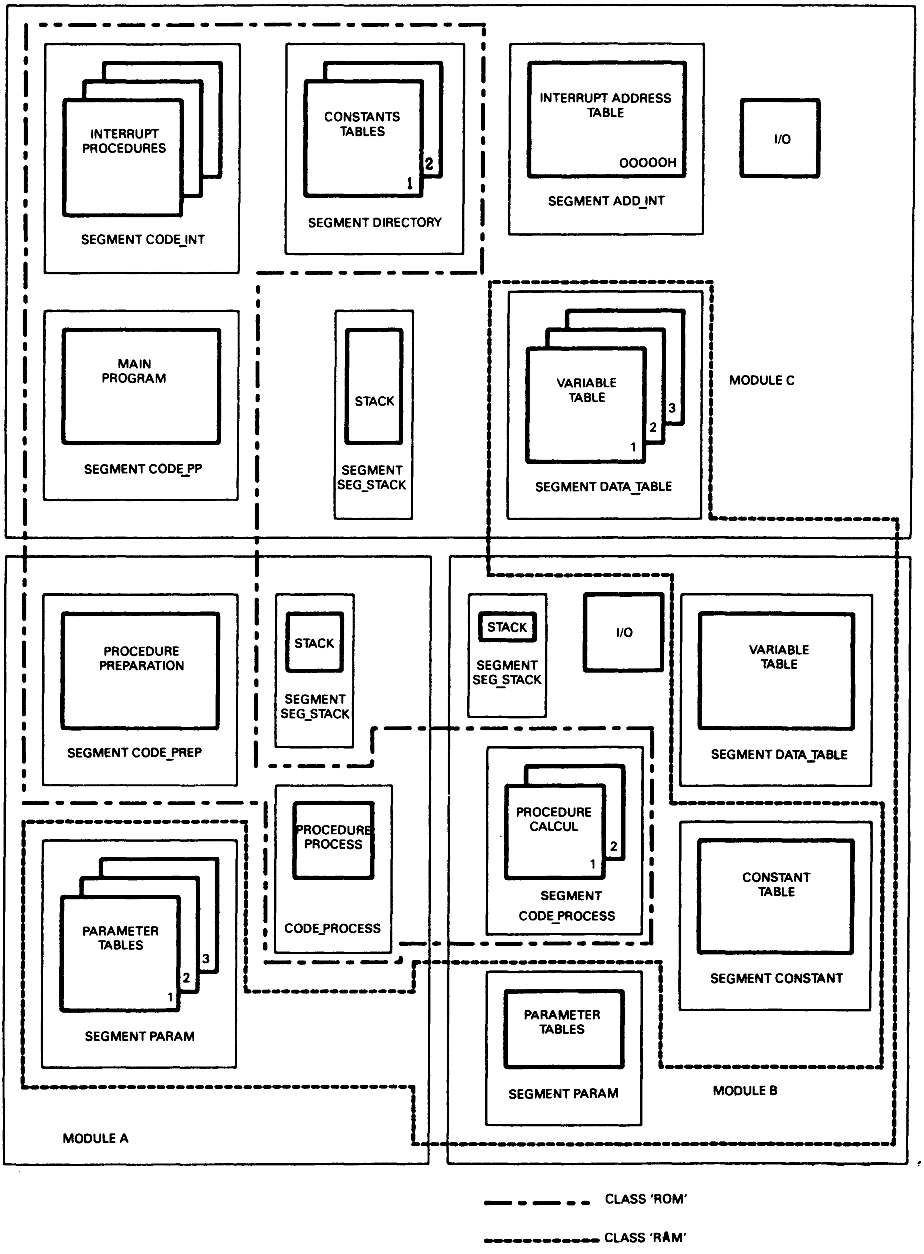

INTERRUPT PROCEDURES
SEGMENT CODE_INT
CONSTANTS TABLES
1
2
SEGMENT DIRECTORY
INTERRUPT ADDRESS TABLE
OOOOOH
SEGMENT ADD_INT
I/O
MAIN PROGRAM
SEGMENT CODE_PP
STACK
SEGMENT SEG_STACK
VARIABLE TABLE
1
2
3
SEGMENT DATA_TABLE
MODULE C
PROCEDURE PREPARATION
SEGMENT CODE_PREP
STACK
SEGMENT SEG_STACK
PROCEDURE PROCESS
CODE_PROCESS
PARAMETER TABLES
1
2
3
SEGMENT PARAM
MODULE A
STACK
SEGMENT SEG_STACK
I/O
VARIABLE TABLE
SEGMENT DATA_TABLE
PROCEDURE CALCUL
1
2
SEGMENT CODE_PROCESS
CONSTANT TABLE
SEGMENT CONSTANT
PARAMETER TABLES
SEGMENT PARAM
MODULE B
CLASS 'ROM'
CLASS 'RAM'

Figure 1.12

2 Program Structures

2.1 Segment

All instructions and variables must be contained within a segment. If no segment is defined, the assembler creates one, called ´??SEG´ which will contain any instructions or variables defined outside of a segment.

2.1.1 Creation of the segment

The SEGMENT and ENDS directives are used to identify the segments as well as to control their alignment, their combination and their contiguity.

Example 2.1

```
;
    CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; ===================================
;           .
;           .
;
    CODE_PP   ENDS
;
              END
```

CODE_PP is the user supplied segment name, to which the assembler assigns the segment value used, as the segment address, in the calculation of the absolute addresses of all the operands used between the lines CODE_PP SEGMENT and CODE_PP ENDS.

BYTE defines the alignment of the segment. The alignment information, which is optional, must follow the SEGMENT directive. The possible aligment directives are as follows.

PARA (Default if no alignment is specified in this field.) The segment begins at an address divisible by 16, that is, one whose last hexadecimal digit is zero (paragraph).

J.-M. Trio, *8086–8088 Architecture and Programming*
© English Language edition, Macmillan Publishers Ltd 1985

BYTE The segment can begin anywhere. In particular, it can ´stick´ to the preceding segment without any intervening space.

WORD The segment begins at an even address.

PAGE The segment begins at an address whose two last hexadecimal numbers are zero. In this case, from 0 to 254 bytes can be left unused between the preceding segment and the present one.

INPAGE (For memory.) All the segment is contained in one page.

The last two alignment directives allow compatibility with the Intel 8-bit microprocessor family.

PUBLIC is a combination directive. There are six such directives.

No indication (Default).
The segment is ´local´ to the module. If a segment with the same name is written in another module, it will be considered to be different.

PUBLIC The segment can be reopened and extended in other modules. The LINK utility looks after this.

AT followed by a value less than 64K.
This directive forces the assembler to align the start of the segment with the address specified.

Therefore, after taking account of the shift applied to all the segment values at the time of the calculation of the physical address, the first physical address of the segment is the value following the AT, multiplied by 16.

Example 2.2

```
;
   PERIPH  SEGMENT  WORD  AT  0E004H
;  ================================
;           .
;           .
;           .
;
   PERIPH  ENDS
;
            END
```

The first physical address of the PERIPH segment, corresponding to a zero offset, is 0E0040H.

This directive is to be used with care because it assumes that the precise memory usage of all other segments is known.

Using this directive places a restriction on the program loader: an overlap of segments may result from this restriction.

On the other hand, the AT directive is a very convenient mechanism for accessing memory-mapped peripherals.

There is no danger of overlap since the LOADER/LOCATER ´knows´ that there is no memory available at this location; it will therefore not load any other segments at this address.

In association with the ORG directive that allows the offset to be fixed (see below), it is thus possible to cause code to be assembled at absolute addresses.

STACK The segment is a stack segment. It can be reopened in other modules, as for the PUBLIC directive.

But in this case only the reserved memory space is increased and the physical address of the last item defined in the segment, at the time of its creation in the main program, is not altered.

This arrangement allows the highest address in the stack segment to be known when the segment is created. The stack is filled from the highest address words down during program execution.

COMMON All segments with the same name that have this combination characteristic are mapped on to the same start address instead of being concatenated.

This provides a COMMON function similar to that of FORTRAN.

MEMORY The segment is loaded at the highest possible address. If several segments have the MEMORY combination, the first segment encountered is loaded high; all other MEMORY segments are then loaded to the same address in the same way as for the COMMON directive.

´ROM´. This is the name of the CLASS to be associated with the segment. The class name must be enclosed within quotation marks.

2.1.2 Generation of the offset in the segment

Location counter

As soon as a segment is created, the assembler allocates a counter to it and sets the counter value to zero. The counter is used to indicate where the next item of information should be written into the segment. Each time data is written into the segment the counter is incremented.

Incrementing stops when the directive ENDS is encountered. It can continue when the segment is reopened in a module or in another module if it has been declared public.

It is even possible to nest segments, although in the interests of readability this practice is not to be encouraged.

Example 2.3

```
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; =========================================
;           .                            => (counter of DATA_TABLE = 0)
;           .
;           .
;           .                            => (counter of DATA_TABLE = 56)
;
  CONSTANT  SEGMENT  WORD  PUBLIC  'ROM'
; ======================================
;           .                            => (counter of CONSTANT = 0)
;           .
;           .
;           .                            => (counter of CONSTANT = 24)
;
   CONSTANT  ENDS
;
;           .                            => (counter of DATA_TABLE = 57)
;           .
;           .
;           .                            => (counter of DATA_TABLE = 74)
;
  DATA_TABLE  ENDS
;
            END
```

Altering the location counter

ORG directive

Example 2.4

```
;
  PERIPH  SEGMENT  WORD  AT  0E004H
; ==================================
;           .                            => (counter = 0)
;           .
;           .                            => (counter = 154 H)
            ORG  1000H
;           .                            => (counter = 1000 H)
;           .
;           .                            => (counter = 1032 H)
;
  PERIPH  ENDS
;
            END
```

On encountering an ORG directive, the value contained in the directive is written into the location counter. When used in a segment that includes an AT combination, it allows the full 20-bit address to be defined completely.

Thus, in the following example, the location of the information following the ORG directive has the address

```
 0E0040 H
+  1000 H
 _________
 0E1040 H
```

EVEN directive

The code or data item following this directive is aligned to an even address, if necessary generating a NOP(NO OPERATION) instruction for the code or modifying the location counter for the data.

Example 2.5

```
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ======================================
;           .                          => (counter = 0)
;           .
;           .
;           .                          => (counter = 52 H)
            EVEN
;           .                          => (counter = 54 H instead of 53 H)
;           .
;
  DATA_TABLE  ENDS
;
            END
```

This directive is only of interest for the 8086 which reads or writes a 16-bit word in a single operation if the word is located at an even address, or in two operations if the word is not aligned to an even address.

It is therefore useful to be able to realign information in this way.

Note that this directive cannot be used in a BYTE type alignment segment.

Direct use of the location counter value

The ´$´ symbol allows reference to the current counter value. It is used in particular for jumps.

Example 2.6

```
;
  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; ====================================
;           .                          => (counter = 0)
;           .
            JMP $ + 06               ; => (counter = 10 H)
;           .
;           .
;           .                          => (counter = 16 H). The  place
                                     ; to  which  the  JMP instruction
                                     ; tranfers control.
;           .                          => (counter = 52 H)
;
  CODE_PP ENDS
;
            END
```

Use of the counter for offset generation

DATA

All variables and constants used within a program are named by the user. The assembler allocates to each name a segment value, an offset and a type. The offset is derived from the value of the counter at the data location.

Example 2.7

```
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ========================================
;             .
;             .
;             .
   VAR_1      DB      05                    ; <= (counter = 23 H)
;             .
;             .
;             .
;
  DATA_TABLE  ENDS
;
              END
```

The segment value of VAR_1 is that of DATA_TABLE; its type is byte (DB directive) and its offset is the value of the location counter (23H), which may be slightly modified to take account of alignment.

As the segment value must be a multiple of 16, if the alignment is not of the PARA type, it is necessary for the assembler to add a small shift to the counter in order to ensure that the segment value is a multiple of 16.

This underlines the danger of wishing to calculate the offset of a variable manually.

For example, it is tempting to assign the value 0 to the offset of the first variable of a byte or word aligned segment. However, this variable can be up to 15 bytes offset from the start of the segment.

It is therefore necessary to use the assembler OFFSET directive to obtain the exact value of the offset of a variable.

CODE

The principle is the same as for data but here the offsets of the labels used when branching have to be defined.

Example 2.8

```
;
  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; ====================================
        ASSUME  CS:CODE_PP
;               .
;               .
;               .
  AGAIN:                              ; <= (counter = 69 H)
;               .
;               .
;               .
                LOOP  AGAIN
;               .
;               .
;               .
;
  CODE_PP  ENDS
;
                END
```

AGAIN is a label whose segment value is that of CODE_PP. Its offset is obtained from the value 69H of the counter. Its type is NEAR or FAR (see next chapter).

LABEL directive

Since the offset is assigned by the position in the text of the variable or the label, how can the same offset be assigned to two variables or to two labels with different names? This dual use of location is required when it is necessary to access the data with differing types, for example, WORD, BYTE.

The LABEL directive permits incrementation of the counter to be suspended so that this problem can be solved.

Example 2.9

```
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ======================================
;             .
;             .
;             .
    ZONE_B      LABEL   BYTE                  ; <= (counter = 23 H)
    ZONE_W      DW      100H                  ; <= (counter = 23 H)
;             .
;             .
;             .
;
  DATA_TABLE  ENDS
;
              END
```

ZONE_B is of type BYTE and ZONE_W is type WORD. Their segment and their offset are identical.

2.1.3 Loading CPU segment registers and pointers

The CPU has four segment registers:

CS: Code segment, used exclusively for program instructions.
DS: Data segment, used for variables and constants.
ES: Extra segment, used for variables and constants.
SS: Stack segment, used exclusively for stack operations.

In addition, there are a further six pointer registers that allow the microprocessor to retain all or part of the offset. They are

IP: Instruction pointer, working exclusively with CS to address program instructions.
SP: Stack pointer, working exclusively with SS.
BP: Base pointer, working in preference with SS.
BX: Base pointer, working in preference with DS.
DI: Destination index, working in preference with DS but necessarily with ES for repeat instructions.
SI: Source index, working in preference with DS.

All these registers have a length of 16 bits.

In an operating system environment, the programmer generally does not have to concern himself with the initialisation of CS, DS and SS if he uses a high-level language to write the main program.

One should take care to preserve these values and restore them to the system on returning from an assembler program, that is, if the assembler alters them. On the other hand, ES may often be available to it.

In a non-operating system environment, the main program must initialise these registers on start up.

Loading and use of CS and IP

These registers are not directly accessible. There are four ways of initialising or altering them.

A RESET

Initialisation of the microprocessor is achieved by pulsing the CPU reset pin.

0FFFFH is then automatically loaded into CS and 0 into IP.

The first instruction executed after RESET is at physical address 0FFFF0H.

An interrupt

After receiving the interrupt number, the CPU looks in the interrupt address table for the offset and then the segment corresponding to the first address of the interrupt subroutine.

It loads them into IP and CS respectively.

A subroutine call: CALL

If this is a FAR type instruction, the values of the offset and segment of the subroutine to be executed are contained in the CALL instruction and loaded into IP and CS.

If it is of the NEAR type, only IP is loaded. However, the subroutine must be located in the same 64K memory block and be placed in the same group as the calling program.

Thus CS is the same for the calling program and for the called program.

A jump: JMP

This instruction operates in the same way as the subroutine call so far as the loading of CS and IP are concerned.

It can also be of FAR or NEAR type. It should however be noted that it is unusual to have to execute a FAR type JMP in a well-structured program.

Conditional jumps

These are all of type NEAR and therefore only alter IP. Furthermore, the branch point must be within +127 or -127 bytes of the jump instruction.

As most conditional jumps only transfer control to an address a short distance away, the designers of the CPU have provided a mechanism whereby the Conditional Jump instruction can be encoded within 16 bits instead of the 24 bits required by the Jump instruction.

The ´short distance´ is defined to be +127 or -127 bytes. Of the 16-bit conditional jump instruction, 8 bits are used to indicate the branch condition, leaving 8 bits to represent a signed offset.

Loading and use of SS and SP

Example 2.10

```
;
  SEG_STACK  SEGMENT  STACK
; ============================
            DW      100H  DUP(?)          ; Before statement, counter = 0.
                                          ; After,counter=200H. Counter will
                                          ; then indicate the next free byte.
   TOP_STACK    LABEL   WORD
;
  SEG_STACK  ENDS
;
  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; =====================================
;           .
;           .
;           .
            MOV     AX, SEG_STACK
            MOV     SS, AX
;           .
;           .
            MOV     SP, OFFSET TOP_STACK
;           .
;           .
            PUSH    AX
;           .
;           .
            POP   AX
;           .
;           .
;
  CODE_PP  ENDS
;
            END
```

The instruction DW 100H DUP(?) reserves 100H (=256) memory words (see next chapter). The variable TOP_STACK thus has 200 H as its offset, and points to the nearest unreserved memory location.

SS is loaded from a data, pointer or index register. In this example, AX is set to the value of the segment name SEG_STACK by using the MOV instruction. A second MOV transfers this value into SS.

Now that the stack segment value is known, the offset value needs to be loaded into SP in order to fix the absolute address of the top of the stack.

SP is initialised, directly this time, by the instruction MOV SP which sets it to 200H.

When it is required to write to the stack, for example by means of the instruction PUSH AX, SP is first decremented by 2, then the absolute address is calculated. When a value is retrieved from the stack using POP AX, SP is incremented by 2 after the absolute address has been calculated and the value extracted.

Assuming that SS is loaded with the value 35H, the example works out as follows:

- before the instruction PUSH AX, SS = 35H and SP = 200H. The absolute location pointed to is 350H + 200H = 550H.
- after the instruction PUSH AX, SS = 35H and SP = IFEH. The value contained in register AX is stored at the physical address 54EH and SP points to this value.
- the address of the value which will be restored to register AX on execution of the POP is calculated from the SS register and the current contents of SP(IFEH); in this case the address is 54EH. The SP is then incremented by 2 (SP = 200H) in order to free this memory location ready for a future PUSH.

This setting of SS and SP must be carried out at the beginning of the main program, either explicitly by the assembler language programmer, or automatically by the use of a high-level language.

With the exception of this initial loading, SS is rarely altered and SP should only be altered with care, otherwise essential data may be lost with catastrophic results, such as the losing of the return address of a subroutine, for example.

The above clearly indicates the use of the STACK alignment directive, because if the STACK_SEG segment is reopened in another module, only the total number of reserved words can be altered and not the highest address (TOP_STACK in our example).

It is in fact essential for the main program to know, on initiation, what the highest address used by the stack is, so that SP can be correctly initialised.

Loading and use of DS, ES and base and index registers

Example 2.11

```
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ====================================
;           .
   TABLE_1    DB      100 DUP(?)
;           .
   TABLE_2    DW      300 DUP(?)
;           .
;
  DATA_TABLE   ENDS
```

```
;
  DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
; =======================================
;              .
  CONSTANT_PI  DD     3.141             ; Warning : MICROSOFT assembler
;              .                          does not follow the IEEE rule
;              .                          for the representation of real
;              .                          numbers (see chapter 3).
;
  DIRECTORY  ENDS
;
  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; =====================================
;              .
               MOV    AX,DIRECTORY
               MOV    DS,AX
;              .
;              .
               MOV    AX,DATA_TABLE
               MOV    ES,AX
;              .
;              .
               MOV    BX,OFFSET TABLE_2
               MOV    DI,150
;              .
;              .
               MOV    SI,OFFSET TABLE_1
;              .
;              .
               MOV    BP,SP
;              .
;              .
;
  CODE_PP   ENDS
;
               END
```

DS and ES are loaded in the same way that SS is loaded; a value is transferred from another register using a MOV instruction. Note that this other register cannot be a segment register. In general AX is used to load DS and ES. On the other hand, the base and index registers are loaded directly, as is shown in example 2.11.

Chapter 4 describes a method whereby a segment and index register can be loaded with a single instruction so as to form a pointer to a particular memory location.

Using specialised index registers like IP and SP does not present any problems over the choice of segment register that will be used in order to reconstruct the absolute address, because these index registers work exclusively with CS and SS respectively.

The same is not true for BP which works implicitly with SS but which can also be linked with DS, ES and CS.

Similarly, BX, DI and SI, which work implicitly with DS, can be linked to SS, ES and CS.

In such circumstances, how does the assembler decide which segment register to use?

Implicit rules

Where no conditions are imposed, the base registers BX and BP are associated with DS and SS respectively.

If used by themselves, indexes DI and SI work with DS. If they are used in conjunction with one of the two base registers, the implicit base register segment predominates.

Example 2.12

```
;
  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; ==================================
;             .
              MOV    AX,[BP]          ; segment= SS and offset= contents of BP
;             .
              MOV    [BX],AX          ; segment= DS and offset= contents of BX
;             .
              MOV    AX,[BX][SI]      ; segment= DS and offset= contents of BX
                                      ;                       + contents of SI
;             .
              MOV    [DI],AX          ; segment= DS and offset= contents of DI
;             .
              MOV    AX,[BP][SI]      ; segment= SS and offset= contents of BP
                                      ;                       + contents of SI
;             .
;
  CODE_PP  ENDS
;
              END
```

Alterations of segment association

In order to override the implicit segment association values, all that is required is to specify a segment prefix, which consists of a segment name followed by a colon, placed in front of the base or index register.

Example 2.13

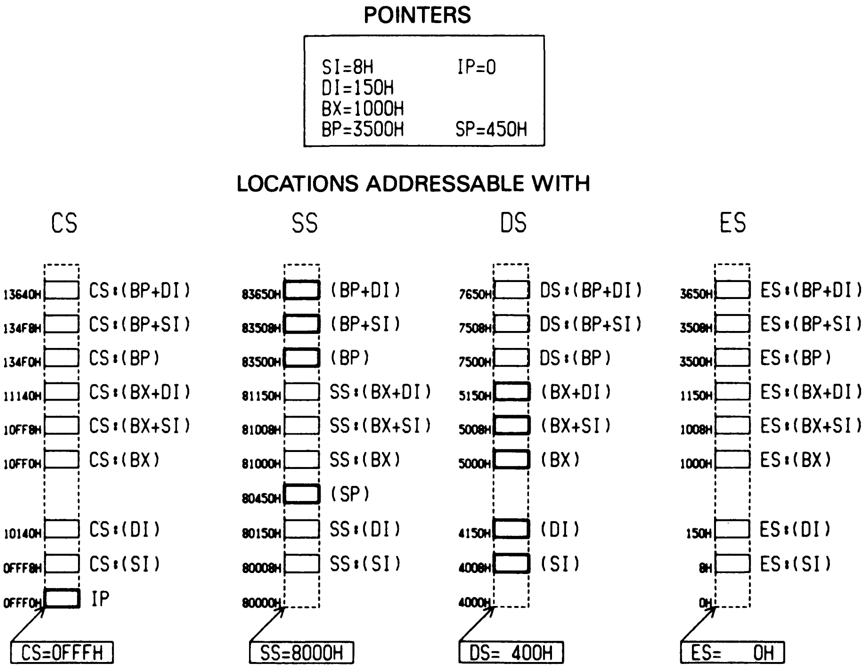
```
;
  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; ==================================
;             .
              MOV    AX,CS:[BP]       ; segment= CS and offset= contents of BP
;             .
              MOV    ES:[BX],AX       ; segment= ES and offset= contents of BX
;             .
              MOV    AX,CS:[BX][SI]   ; segment= CS and offset= contents of BX
                                      ;                       + contents of SI
;             .
              MOV    SS:[DI],AX       ; segment= SS and offset= contents of DI
;             .
              MOV    AX,DS:[BP][SI]   ; segment= DS and offset= contents of BP
                                      ;                       + contents of SI
;             .
;
  CODE_PP  ENDS
;
              END
```

After example 2.13 has been assembled, an investigation of the listing generated by the assembler shows the segment prefix appearing in the columns of generated code. This prefix makes the CPU choose a segment register different from the implicit one.

	Prefix
CS	2E H
PS	3E H
ES	26 H
SS	36 H

Figure 2.1

Figure 2.1 shows which absolute addresses the CPU can reach, given the values of the segment and index registers shown.


POINTERS
SI=8H
IP=0
DI=150H
BX=1000H
BP=3500H
SP=450H
LOCATIONS ADDRESSABLE WITH
CS
SS
DS
ES
13640H CS:(BP+DI)
134F8H CS:(BP+SI)
134F0H CS:(BP)
11140H CS:(BX+DI)
10FF8H CS:(BX+SI)
10FF0H CS:(BX)
10140H CS:(DI)
0FFF8H CS:(SI)
0FFF0H IP
CS=0FFFH
83650H (BP+DI)
83508H (BP+SI)
83500H (BP)
81150H SS:(BX+DI)
81008H SS:(BX+SI)
81000H SS:(BX)
80450H (SP)
80150H SS:(DI)
80008H SS:(SI)
80000H
SS=8000H
7650H DS:(BP+DI)
7508H DS:(BP+SI)
7500H DS:(BP)
5150H (BX+DI)
5008H (BX+SI)
5000H (BX)
4150H (DI)
4008H (SI)
4000H
DS= 400H
3650H ES:(BP+DI)
3508H ES:(BP+SI)
3500H ES:(BP)
1150H ES:(BX+DI)
1008H ES:(BX+SI)
1000H ES:(BX)
150H ES:(DI)
8H ES:(SI)
0H
ES= 0H

2.2 The PROC/ENDP Directive

Example 2.14

```
  CODE_PREP   SEGMENT   BYTE   PUBLIC   'ROM'
; =======================================
         ASSUME          CS:CODE_PREP
```

```
;
 PREPARATION   PROC                           ; equivalent to PROC NEAR
; ----------------
;             .
;             .
;             .
              RET
;
 PREPARATION   ENDP
;
  CODE_PREP   ENDS
;
              END
```

Example 2.15

```
;
  CODE_INT  SEGMENT  BYTE  PUBLIC  'ROM'
; ====================================
;
 INT_1  PROC  FAR
; --------------
;             .
;             .
              IRET
;
 INT_1  ENDP
;
  CODE_INT   ENDS
;
              END
```

A procedure is implicitly of type NEAR. It maybe specified as type FAR.

In the case of a NEAR procedure, only the return address offset is saved at the call and restored at the return (RET). It is therefore necessary for the procedure to be in the same segment or in the same group as the calling program (example 2.14).

In the case of a FAR procedure, the offset and the return address segment are saved, and are restored on the execution of a RET instruction. There are no memory constraints for this procedure.

The FAR type should be used whenever a procedure written in assembler is called by a high level language or when an interrupt procedure is involved. In the latter case, IRET should be used instead of RET (example 2.15), because the status word is also saved when an interrupt occurs. The IRET instruction resets the status word from the stack (see chapter 5 on interrupts).

2.3 GROUP Directive

The GROUP directive specifies that certain segments belong to one and the same block of 64K of memory and that it is possible to assign to them a common segment value for the calculation of absolute addresses.

Example 2.16

```
    CODE        GROUP   CODE_PP,CODE_PREP,CODE_PROCESS
;
   CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
;  ==============================
        ASSUME  CS:CODE
;               .
;               .
                CALL    PREPARATION
;               .
;               .
   CODE_PP   ENDS

  CODE_PREP   SEGMENT   BYTE   PUBLIC   'ROM'
;  ================================
        ASSUME  CS:CODE
;
  PREPARATION  PROC  NEAR
;  ----------------------
;               .
;               .
                CALL    CALCUL_1
;               .
   TWO          DW      2
;               .
;               .
  PREPARATION  ENDP
;
  CODE_PREP   ENDS

  CODE_PROCESS   SEGMENT   BYTE   PUBLIC   'ROM'
;  ==================================
        ASSUME  CS:CODE
;
  CALCUL_1  PROC  NEAR
;  -------------------
;               .
;               .
                MOV     BX,OFFSET CODE:TWO
;               .
;               .
                MOV     AX,CS:[BX]
;               .
;               .
  CALCUL_1  ENDP
;
  CODE_PROCESS   ENDS
;
                END
```

We will take the following values for the symbols used in example 2.16.

CODE = 200H
CODE_PREP = 315H
CODE_PROCESS = 520H

TWO = offset in CODE_PREP = 63H
PREPARATION = offset in CODE_PREP = 7H
CALCUL_1 = offset in CODE_PROCESS = 15H

The group name CODE represents the same value as the first segment in the list CODE_PP:

CODE = CODE_PP = 200H

On entering the main program in the segment CODE_PP, CS is set to the value 200H and IP points to each instruction in turn.

With reference to the subroutine call of PREPARATION, since PREPARATION has been declared NEAR, although it is not in the segment CODE_PP, the assembler must alter the offset of PREPARATION in order to generate the correct code.

Using the values given, the absolute location of the first instruction of PREPARATION is

3150H + 7H = 3157H

As CS is equal to 200H when the subroutine is called, if IP is simply made equal to the offset of PREPARATION, 7H, the next instruction address would be 2000H + 7H = 2007H which is incorrect.

What actually occurs is that the assembler recognises that a different segment is involved. It then looks to see which group this segment is in and adds to the offset of PREPARATION the distance between CODE_PREP and CODE, multiplied by 16, which gives

CODE_PREP - CODE = 315H - 200H = 115H.

The actual value loaded into IP is

OFFSET PREPARATION + (CODE_PREP - CODE) * 16
= 7H + 1150H = 1157

The next instruction address, using CS = 200H, is then

2000H + 1157H = 3157H (correct).

From the above example, it can be seen that if the GROUP is not restricted to a maximum size of 64K bytes, the assembler would be unable to generate these modified 16-bit offsets.

For the CALCUL_1 call, the calculation is as follows

absolute address = CODE_PROCESS:CALCUL_1 = 5200H + 15H
= 5215H
CS = 200H -> 2000H
IP = 15H + (520H - 200H) * 16 -> 3215H

Absolute address = 5215H
(correct)

In fact the assembler has to be told that it needs to work with the group rather than with segments.

This is the role of the ASSUME directive which works in the following way.

Let us assume that the assembler has now reached the PREPARATION subroutine call.

Now this subroutine is in the CODE_PREP segment which uses the ASSUME directive to set CS to the value of the GROUP instead of the segment itself.

The assembler therefore modifies all the offsets accordingly.

The ASSUME directive is described in greater detail in chapter 4.

Finally, in our example, if BX is loaded with the offset of ´TWO´ which has the value 63H in CODE_PREP, the assembler will need to be informed that it must correct this value.

This operation is carried out by writing

```
MOV BX, OFFSET CODE:TWO
```

In this case, OFFSET CODE:TWO has the value

63H + (315H - 200H) * 16 = 63H + 1150 H = 11B3H

and this value is loaded into BX. Consequently, on the instruction

```
MOV AX, CS:[BX]
```

the absolute operand address is 200H : 11B3H = 2000H + 11B3H = 31B3H pointing correctly to the word containing the value 2.

The instruction LEA BX, TWO (Load Effective Address) carries out the same operation, using a simpler form of coding, because it interrogates the ASSUME directive in order to establish to which group the operand belongs.

2.4 The Link Between Modules

In order to allow symbolic references from one module to another, the assembler provides three directives: NAME, PUBLIC and EXTRN. The use of each directive is described below.

2.4.1 NAME directive

In each module, the NAME directive, which has the syntax NAME name, allows the user to associate a name with each module. For the user, this name only reappears in the MAP file obtained after executing the utility program LINK.

2.4.2 PUBLIC directive

All the variables or subroutines used by other modules must appear in a PUBLIC directive. If not, they are considered to be local to the module in which they are defined and invisible to other modules. While this allows the same name to be used in different modules without confusion, it is not recommended.

2.4.3 EXTRN directive

The EXTRN directive indicates to the assembler the name and type (and possibly the segment name) of a symbol defined within another module.

The type can be literally

BYTE, WORD, DWORD	for variables
NEAR, FAR	for code
ABS	for numbers

The types QWORD, TBYTE, STRUCTURE or RECORD are only available in the latest versions of the Intel assembler and are not dealt with here. In order to specify the segment name, the EXTRN directive must be declared within a segment of the correct name. The segment is specially reopened for this if necessary.

The variable may then be used like a local variable. This is the case with the variable GLOBAL in example 2.17.

If the segment is not known, it is possible to leave it to the assembler to find it by placing the directive EXTRN outside all the segments (as in the case of VAR_EXT in example 2.17) and by using the directive SEG to load a segment register, generally ES.

Finally, the prefix ES: allows us to specify to the assembler which segment register should be used with this variable to calculate the absolute address.

Example 2.17

```
            NAME    MODULE_A
;           ================

            PUBLIC  GLOBAL

            CODE    GROUP  CODE_PP,CODE_PREP
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ======================================
   GLOBAL      DW      1035H
;
  DATA_TABLE  ENDS

  CODE_PREP  SEGMENT  BYTE  PUBLIC  'ROM'
; =====================================
            EXTRN   PREPARATION:NEAR
;
  CODE_PREP  ENDS
  CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; ==========================================
      ASSUME   CS:CODE                        ; ASSUME: see chapter 4.

MAIN_PROG_START:
;             .
              CALL   PREPARATION
;             .

  CODE_PP    ENDS
;
              END   MAIN_PROG_START


              NAME    MODULE_B
;             ===============

              PUBLIC  PREPARATION
              EXTRN   VAR_EXT:WORD

              CODE    GROUP CODE_PP,CODE_PREP
;
  DATA_TABLE   SEGMENT   WORD   PUBLIC   'RAM'
; ======================================
              EXTRN   GLOBAL:WORD
;
  DATA_TABLE   ENDS

  CODE_PREP   SEGMENT   BYTE   PUBLIC   'ROM'
; ====================================
      ASSUME  CS:CODE,DS:DATA_TABLE           ; ASSUME: see chapter 4.
;
 PREPARATION  PROC  NEAR
; -------------------
;             .
              MOV     AX,SEG VAR_EXT          ; Load ES with the  segment  con_
                                              ; taining VAR_EXT.
              MOV     ES,AX
;             .
              MOV     AX,GLOBAL
              MOV     ES:VAR_EXT, BX          ; Assembler  does  not  know  the
                                              ; segment of VAR_EXT, nor the re_
                                              ; gister segment  concerned. This
                                              ; must be indicated to the assem_
                                              ; bler.
 PREPARATION  ENDP
;
  CODE_PREP   ENDS
;
              END
```

2.4.4 END directive

END tells the assembler that the source of program is terminated.

In a module, there should be only one END directive and this should be the last statement. (Of course, this must be followed by a carriage return.)

In the module that contains the main program, and uniquely in this case, the END directive should be followed by the start address of the main program (see in example 2.17 the end of module A).

This is the address to which the loader transfers control after the program has been loaded.

3 Definition and Initialisation of Data

3.1 Writing Identifiers

An identifier is a name attributed to an item of data, a segment, a procedure, and so on. It possesses the following characteristics.

it begins with a letter or one of the three special characters ? @ or -

it can contain letters, names or the three special characters

it can contain up to 31 characters.

Over and above this number the characters are ignored, even though they still appear in the .LST file.

3.2 Identifier Attributes

The assembler recognises three forms of identifier

Constants: pure numbers, without attribute

Variables: data that can be manipulated. They are the operands of the instructions MOV, ADD, and so on.

Labels: point to the program instructions. They are the operands of the CALL, JMP and conditional jump instructions.

Variables and labels have different attributes, depending on the following.

What is the physical location?
   What is the segment value?
   What is the offset value?

How is this variable or label used?
   What is the type of a variable, counted in bytes?

```
BYTE           TYPE =  1
WORD           TYPE =  2
DWORD          TYPE =  4
QWORD          TYPE =  8
TBYTE          TYPE = 10
Structure name TYPE =  n (number of bytes in
                          structure)
```

J.-M. Trio, *8086–8088 Architecture and Programming*
© English Language edition, Macmillan Publishers Ltd 1985

What is the type of a label

NEAR
FAR

3.3 Constants

Constants are pure numbers presented to the assembler either in direct form in different bases, or in symbolic form specified using the directive EQU prior to the first use of the symbol.

3.3.1 Use in direct form in an instruction

The assembler translates the values, written in different bases, into two´s complement binary occupying 16 bits (giving a signed range of -32768 to +32767 or an unsigned range of 0 to 65536). The number handled can be the result of an arithmetic operation +, -, *, /, MOD or a logical operation AND, OR, XOR, NOT.

Example 3.1

```
  CODE_PROCESS  SEGMENT  BYTE  PUBLIC  'ROM'
; ========================================
      ASSUME  CS:CODE_PROCESS              ; ASSUME: see chapter 4.
;
 CALCUL_2  PROC  NEAR
; -------------------
;                 .
                  MOV     AX,15            ; Decimal value: 0F H stored in AX
;                 .
                  MOV     BX,-12D          ; Decimal value:0FFF4 H stored in BX
;                 .
                  MOV     DI,77Q           ; Octal value: 3F H stored in DI
;                 .
                  MOV     SI,101B AND 100B ; Binary value: 04 H stored in SI
;                 .
                  MOV     BP,'A'           ; ASCII value: 41 H stored in BP
;                 .
                  MOV     CX,25H*2H        ; Hexadecimal value:4A H stored in CX
;                 .
                  MOV     AX,'8' AND 0FH   ; 38 H AND 0F H: 08 H stored in AX
;                 .
                  MOV     CX,100 MOD 18H   ; modulo : 04 H stored in CX
;
 CALCUL_2  ENDP
;
  CODE_PROCESS  ENDS
;
                  END
```

3.3.2 EQU directive

Figure 3.2 shows the use of the EQU directive to form a symbolic constant; it also shows hows the symbolic constant is used.

Example 3.2

```
     SIXTEEN EQU     54*4-200
;
  CODE_PROCESS  SEGMENT  BYTE  PUBLIC  'ROM'
; =========================================
     ASSUME  CS:CODE_PROCESS              ; ASSUME: see chapter 4.
;
  CALCUL_2  PROC  NEAR
; ----------------------
;            .
             MOV    AX,SIXTEEN            ; Symbolic value 10 H stored in AX
;            .
;            .
  CALCUL_2  ENDP
;
  CODE_PROCESS  ENDS
;
             END
```

The following section describes the use of constants to initialise variables.

3.4 Definition of Variables

A variable is completely defined when the assembler knows its segment, its offset, its type and possibly its initial value, which may remain undefined as long as it is not used by the program during execution.

Segment and offset were examined in chapter 2. This section therefore deals with the type and initial value.

3.4.1 Initialisation with a constant expression

Example 3.3

```
  DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
; ====================================

   TABLE_1          LABEL   BYTE

   SERIAL_NUMBER    DB      23H + 5
   IDENTITY         DB      'MR SMITH'
   NUMBER           DW      1234H
   REDUCTION        DD      0.80          ; Warning :  MICROSOFT  assembler
                                          ; does not  follow  the  rule  of
                                          ; numeric representation (see  DD
                                          ; in this chapter).
   NUMERATOR        DW      80
   DENOMINATOR      DW      100
   PRECISION        DQ      18446744073709551615
   LARGE_NUMBER     DT      12345678901234567B
   COUNTY           DW      '67'

   END_TABLE_1      LABEL   BYTE
;
  DIRECTORY  ENDS
;
                    END
```

The method of initialising a variable depends upon the type of the variable. The use of the keywords DB, DW, DD, DQ, DT is described below.

DB (Define Byte)

DB accepts all signed values lying between -128 and +127; it also accepts all unsigned values between 0 and 255.

For numerical values only a single byte is initialised. Additionally, DB is used to initialise strings of more than 2 characters. This is demonstrated with the initialisation of IDENTITY in example 3.3.

In this case, 8 bytes have been allocated and loaded with the ASCII values, the ´M´ at the lowest address, the ´H´ at the highest address.

If a quote mark (´) is included in the character string, it must be preceded by an additional single quote - thus (´´). For Microsoft assembler, it must be included in its ASCII form, that is, 39 or 27H. The string length is limited to 255 characters.

DW (Define Word)

DW allocates 2 bytes and accepts all signed values lying between -32768 and +32767, and unsigned values in the range 0 to 65536.

It should be noted that the lower part of the number is loaded at the lowest address and the higher part at the highest address. Thus, in example 3.3, 34H is loaded at the offset NUMBER and 12H at the offset NUMBER + 1.

It is possible to load a word with two characters, as for the variable COUNTY, but here the ASCII code for 6 is loaded at the offset COUNTY + 1 and the ASCII code for 7 at COUNTY, in contrast to loading with DB.

DD (Define Double Word)

Use of DD allows 2 words or 4 bytes to be allocated.

The possible numerical values are as follows

(a) an integer from -2^{31} to $+2^{31} -1$

That is, -2 147 483 647 to 2 147 483 646.

(b) a real number

from -2^{-128} to -2^{-126}, 0, 2^{-126} to 2^{128}

covering the decimal values from +-3.37 E+38 to +-1.18 E-38 (simple precision in FORTRAN) with an empty interval either side of zero.

(c) an assembler expression in 16 bits with the most significant bits completed by 0´s or 1´s according to the sign bit,

(d) loading of a string of a maximum of two characters, carried out as for DW at the low address word, with 0 being loaded in the high address word (wrong usage).

Special note for Microsoft assembler users

The numeric representation of Microsoft´s 1.06 version does not conform to the IEEE standard: the sign is placed at the beginning of the mantissa instead of at the beginning of the number and the exponent shift is not the same.

This fact must be taken into account when transferring arguments with high-level languages and for the defining of constants used by the 8087. In the latter case, it is preferable to define such constants with integer values.

Thus in example 3.3., the instruction

```
   FLD   REDUCTION
```

should be replaced by the following lines which load the value 0.80

```
   FILD  REDUCTION_NUM
   FIDIV REDUCTION_DENOM
```

Here the first instruction loads the integer value 80, while the second divides this value by 100.

The Microsoft representation can also be converted to the IEEE form, and vice versa, as is shown in example 4.21a.

DQ (Define Quadword)

DQ allocates 8 bytes to the following numerical values

INTEGER: from -2^{63} to $+2^{63} -1$

That is, approximately, the values below 9.22 x 10^{18}, in absolute value.

REAL: from -2^{1024} to -2^{-1022}, 0, $+2^{-1022}$ to 2^{1024} with 2^{1024} = 1.7 E + 308 covering the decimal values from +-1.80 10^{308} to +-2.23 10^{-308} (double precision in FORTRAN).

an assembler expression in 16 bits completed by 0´s or 1´s depending on the sign bit.

loading a string of two characters carried out under the same conditions as for DW, the more significant words being initialised with zeros.

Microsoft assembler users should note that in the numerical representation of real numbers the same non-standard features apply, as mentioned above under the DD directive.

Note that variables defined with the aid of DQ can only be referenced in EXTRN in the latest versions of the assembler.

DT (Define Ten bytes)

The allocation of 10 bytes is made for operation with the 8087 mathematics co-processor (see chapter 7), either to output or input magnitudes presented as a string of BCD numbers ranging from

$$- (10^{-18} -1) \text{ to } + (10^{+18} -1)$$

or to write and re-read numbers in the internal format of the 8087.

In fact the 8087 mathematics co-processor carries out all its operations with numbers lying between

$$-2^{16384} \text{ to } -2^{-16382}, 0, 2^{-16382} \text{ to } 2^{16384}$$

$$\text{with } 2^{16384} = 1.2 \times 10^{4932}$$

This format is only useful in memory to store intermediary calculation results.

DT can also be used to initialise a two-character string, subject to the same rules as for DD and DQ.

Note that, as for DQ, DT can only be used to define a variable referenced in EXTRN in recent versions of the assembler.

3.4.2 Variable definition without initialisation

A question mark replaces the numerical values in all the types so far described.

Only the required number of bytes are reserved, with no initialisation taking place.

3.4.3 Pointer or index definition

A pointer or an index is a variable containing the segment and offset or offset alone of an instruction label or of the address of another variable.

Example 3.4

```
                DATA    GROUP  DATA_TABLE,DIRECTORY
;
  DIRECTORY  SEGMENT  WORD  PUBLIC
; ================================
   TABLE_1         LABEL    BYTE
;                  .
;                  .
   END_TABLE_1     LABEL    BYTE

   ADD_TABLE_1     DD       TABLE_1
;
  DIRECTORY  ENDS

  CODE_PROCESS  SEGMENT  BYTE  PUBLIC  'ROM'
; ===========================================
       ASSUME  CS:CODE_PROCESS,DS:DIRECTORY     ; ASSUME: see chapter 4.
;
  CALCUL_3  PROC  FAR
; -----------------
;                  .
                   LDS      BX,ADD_TABLE_1
;                  .
                   IRET
;
  CALCUL_3  ENDP
;
  CODE_PROCESS  ENDS

  ADD_INT  SEGMENT  WORD  'RAM'
; ============================
   INT_CALCUL_3    DD       CALCUL_3
;                  .
;                  .

  ADD_INT  ENDS

  DATA_TABLE  SEGMENT  WORD  'RAM'
; ================================
;                  .
   LONG            DW       END_TABLE_1-TABLE_1
   DEF_OFFSET_1    DW       TABLE_1
   DEF_OFFSET_2    DW       OFFSET TABLE_1
   DEF_OFFSET_3    DW       OFFSET DATA:TABLE_1
;                  .

  DATA_TABLE  ENDS
;
                   END
```

Pointers

In a pointer, the offset is always in the low address word and the segment in the high address word.

The instruction LDS BX, ADD_TABLE_1 in example 3.4 illustrates the use of such a pointer. The segment stored at the offset ADD_TABLE_1 +2 is loaded into DS while the offset stored at ADD_TABLE_1 is loaded into BX.

There is also a LES instruction which operates in the same say with the ES register. (See the instruction set or example 4.11.)

This method of working, by pointer, is often used for passing parameters that come from a program written FORTRAN or Pascal. In this case, the pointer is read in from the stack (see chapter 4). The pointers are also frequently used to establish the interrupt vector table, illustrated in example 3.4 by the pointer INT_CALCUL_3 which contains the branch address to the interrupt program CALCUL_3.

Indexes

The definition and use of offsets is not entirely straightforward. The problem is demonstrated in example 3.4 by the three index definitions DEF_OFFSET_1, DEF_OFFSET_2 and DEF_OFFSET_3.

The first possibility is to load DEF_OFFSET_1 with the offset value of TABLE_1 calculated at the time of the assembly of the module without taking account of other information being added to the segment in other modules.

In all probability the resultant offset will be incorrect.

The correct use of variable names for the manipulation of offsets at assembly time is shown by the variable LONG loaded with a number representing the difference of two offsets. Their absolute value is then no longer of importance.

Incidentally, this usage is very convenient since it allows TABLE_1 to be lengthened without requiring major changes in the rest of the program.

The second definition DEF_OFFSET_2 takes account, for the loading of the offset, of its value in the segment after LINK has linked all the modules. It thereby takes into account any other use of the segment in other modules.

However, if access to TABLE_1 is made with the aid of a segment register loaded with the DATA value, base of the group containing DIRECTORY, the calculation of the address will again be incorrect.

In fact, account must be taken of the shift that exists between the base of the DIRECTORY segment that contains TABLE_1 and the base of the DATA group.

The third definition, DEF_OFFSET_3, is initialised with the offset of TABLE_1 in DIRECTORY, after LINK, to which is added the distance (DIRECTORY-DATA) * 16.

DEF_OFFSET_2 and DEF_OFFSET_3 are two possible choices of offset calculation. One or other must be indicated to the assembler as a function of the loading of the segment register used for the calculation of the physical address of TABLE_1.

Now this loading is not known to the assembler, since it is made at the time of program execution and the actual position of the instruction that causes this loading is perhaps contained in another module, for example, in the module containing the main program.

Again, it is necessary to specify for the assembler, the value of the segment registers at the time of execution. The directive ASSUME allows the assembler to verify the choice of offset or select the correct segment register when it can do it itself.

This directive is studied in chapter 4.

3.4.4 Definition and initialisation of a data list

A single variable definition can point to a list of data values stored one after the other.

Initialisation of a string of characters by the directive DB already operates in this way.

It is possible to extend this concept to all types, provided that, for each directive, no more than 16 successive data values are specified.

Thus, a single DB directive can support up to 16 strings, each of 255 characters.

Example 3.5

```
   DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
;  ========================================
;              .
    PRIME_NOS  DW     2,3,5,7,11,13,17,19
    IDENT      DB     'FAMILY NAME',0DH,0AH,'NAME',0DH,0AH
    TABLE      DD     1.0,2.0,3.0,4.0,5.0,6.0
               DD     2.0,4.0,6.0,8.0,10.0,12.0
               DD     2.0,4.0,8.0,16.0,32.0,64.0
;              .

   DIRECTORY  ENDS
;
               END
```

3.4.5 Values repeated several times

The DUP symbol allows a data value, or a simple memory allocation without initialisation, to be repeated a precise number of times.

Example 3.6

```
   DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
;  =====================================
   VAR_0      DB     100 DUP(0)
              DB     2 DUP((0),3 DUP(4))          ; => 0,4,4,4,0,4,4,4
   ERROR      DB     100 DUP('ERROR',0DH,0AH)     ; => 100 lines 'ERROR'
   NUMBERS    DW     100 DUP(5 DUP(5),7)          ; => 100 copies 5,5,5,5,5,7
              DT     100 DUP(0)                   ; => 1000 bytes of 0
              DQ     50 DUP(7)                    ; => 400 bytes reserved
                                                  ; with initial value of 7
;
```

```
  DIRECTORY  ENDS
;
             END
```

Note that DUP is also used to reserve one or more bytes without initialisation.

```
WORD_1   DB   2 DUP(?)
WORD_2   DW   1 DUP(?)
```

In this case no initialisation is generated, whereas

```
WORD_1   DW ?
WORD_2   DW ?
```

depending on the systems, can set these locations to an inderminate value. This can cause problems if the locations are those of the registers of a memory-mapped peripheral.

3.5 RECORD

3.5.1 Definition

The directive RECORD allows a new type to be defined, which is then used in the same manner as DB or DW, and which allows bit ´fields´ to be defined within a byte or a word.

The fields are defined by means of a symbol, followed by a colon and then by the number of bits in the field.

If the total number of bits defined is less than 8 for a byte and 16 for a word, the bits involved will be the least significant bits.

When the type is being defined, each field can be initialised by adding an equals sign followed by a value after the number of bits in the field.

If no value is given, an initial value of zero is assumed.

Finally, if the field is exactly 8 bits wide, a character can be used to initialise the field.

3.5.2 Using RECORD for the definition and initialisation of variables

Example 3.7

```
                NAME    MODULE_A
;               ================

                FUBLIC  TABLE_B

    MODEL_1     RECORD  X:3=7, Y:4=8, Z:9=257
    MODEL_2     RECORD  NUMBER:4=0FH, FIGURE:8='3'
```

```
;            Initial format of MODEL_1:
;           15  13 12     9 8                0
;           --------------------------------
;            1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1
;           --------------------------------
;            < X > <  Y  > <       Z        >

;            Initial format of MODEL_2:
;                  11     8 7                0
;           --------------------------------
;            X X X X 0 1 0 0 0 0 1 1 0 0 1 1
;           --------------------------------
;                  <NUMBER> <    FIGURE     >


;
  DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
; ==================================
;                .
   TABLE_B       MODEL_1       < >
                 MODEL_1       5 DUP(<,15,3>)
                 MODEL_1       5 DUP(<,,1>)
   STATUS_WORD   MODEL_2       < >
                 MODEL_2       <5,'9'>
;                .

  DIRECTORY  ENDS

  CODE  SEGMENT  BYTE  PUBLIC  'ROM'
; ==============================
;            .
             MOV     AX,DIRECTORY
             MOV     ES,AX
;            .
;            .
             MOV     BX,OFFSET TABLE_B
             MOV     DI,0
             MOV     AX,ES:[BX][DI]            ; 1111000100000001 B stored in AX
;            .
;            .
             MOV     DI,5*2
             MOV     AX,ES:[BX][DI]            ; 1111111000000011 B stored in AX

;            .
;            .
             MOV     DI,7*2
             MOV     AX,ES:[BX][DI]            ; 1111000000000001 B stored in AX
             MOV     AX,ES:STATUS_WORD         ; 0000111100110011 B stored in AX
             MOV     AX,ES:STATUS_WORD[2]      ; 0000010100111001 B stored in AX
;
  CODE  ENDS
;
             END
```

3.5.3 Use for the calculation of a constant expression

One of the main uses of record structuring at the bit level is the ease with which bits can be read or written in bytes or words associated with a peripheral or I/O port.

This generally involves programmable integrated circuits that require bit-structured control words (see chapter 6 and following). It is therefore practicable to load the fields as though a 1 byte or 1 word variable was involved. To this end, it is also possible to use directly the type defined in RECORD in an instruction operand or in an EQU in order to calculate a constant expression.

Example 3.8

```
              NAME    MODULE_A
;             ================

   R          RECORD  D3:2, D2:6, D1:2, D0:6
   NUMBER_BCD EQU     R<1,2,3,4>
;
  CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; ====================================
;             .
              MOV     AX,NUMBER_BCD           ; 0100001011000100B stored in AX
;             .
              MOV     BX,R<,8,,20>            ; 0000100000010100B stored in BX
;             .
              MOV     DX,0AF75H AND R<3,0,3,0>   ; 1000000001000000B stored in DX
;             .

  CODE_PP   ENDS
;
              END
```

3.5.4 Operators associated with RECORD

The type RECORD is also used to read bit fields within a WORD or a byte. Here again, the case of reading from a peripheral, such as a port, is typical.

In fact, the external data is generally present on one or more lines of a port. In order to isolate the data of interest it is first necessary to isolate the field. This is done with the aid of the operator MASK.

Then, the result is shifted so that normal formatting takes place. For this the name of the field itself gives the necessary shift number. Finally, when the data gathered in this way is processed, it is often practicable to use the number of bits of a field, given by the operator WIDTH.

Example 3.9

```
              NAME    MODULE_A
;             ================

      R       RECORD  D3:2, INPUT_2:6, D1:2, D0:6
;
  CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; ====================================
      ASSUME  CS:CODE_PP                ; ASSUME: see chapter 4.
;             .
              MOV     DX,0AE75H         ; 1010111001110101 B stored in DX
              AND     DX,MASK INPUT_2   ; MASK INPUT_2 = 0011111100000000 B
                                        ; 0010111000000000 B stored in DX
;             .
```

```
            MOV     CL,INPUT_2          ; 8 stored in CL
            SHR     DX,CL               ; Shift right to "CL" bits
                                        ; 0000000000101110 B stored in DX
                                        ; DX = 2E H
;           .
            MOV     CX,WIDTH INPUT_2    ; 6 stored in CX
   NEXT_BIT:                            ; The loop  starting at  NEXT_BIT  is
                                        ; repeated as many times as the  num_
                                        ; ber of bits in the field.
;           .
            LOOP    NEXT_BIT
;
  CODE_PP   ENDS
;
            END
```

3.6 Structure

Just as RECORD allows new ´personalised´ types to be defined, partitioning the data at the bit level, so STRUC makes it possible to define new types of structured variables according to any combination of any other type.

It is really a question of defining a ´super type´.

3.6.1 Definition and use in reserving memory

Example 3.10

```
            NAME     MODULE_A
;           ================

   IDENT_NUMBER    STRUC
;  - - - - - - - - - - -
   SEX          DB      ?
   YEAR         DW      ?
   MONTH        DW      ?
   STATE        DW      ?
   NUMBER       DW      ?
                DW      ?
                DW      ?
;
   IDENT_NUMBER     ENDS                ; Length of IDENT_NUMBER=13 bytes

   IDENT_CLIENT    STRUC
;  - - - - - - - - - - -
   FAMILY_NAME DB      15 DUP(?)
   NAME        DB      15 DUP(?)
   STREET_NUM  DW      ?
   STREET      DB      25 DUP(?)
   CITY        DB      25 DUP(?)
   POSTAL_CODE DB      5  DUP(?)
;
   IDENT_CLIENT     ENDS                ; Length of IDENT_CLIENT=87 bytes
```

```
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; =====================================

FILE_CLIENT     IDENT_CLIENT   100 DUP(< >)  ; Dimension of FILE_CLIENT=87*100
                                             ; = 8700 bytes
FILE_NUM        IDENT_NUMBER   100 DUP(< >)  ; Dimension of FILE_NUM=13*100
                                             ; = 1300 bytes
;                 .
  DATA_TABLE  ENDS
;
                END
```

For the assembler, the value associated with TYPE is the number of bytes in the structure. In example 3.10, TYPE IDENT_CLIENT = 87.

3.6.2 Use with initialisation and initialisation overwriting

When a type is defined with the aid of STRUC, the fields can be initialised to a given value, which is used when a variable of this type is created.

When this variable is created, it is possible to impose a new initial value provided that the field does not consist of several parts (see example 3.12).

Example 3.11

```
                NAME   MODULE_A
;               ===============


;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; =====================================
   ADDRESS_1    LABEL   WORD
   COPY_1       MODEL   < >
   COPY_2       MODEL   <,0,,,,255>
   COPY_3       MODEL   3 DUP(<,,,,50>)
;
  DATA_TABLE  ENDS
;
   MODEL        STRUC
; - - - - - - - - -
   ONE          DW      0FFEH
   TWO          DW      OFFSET ADDRESS_1
   THREE        DB      7,5
   FOUR         DB      'A'
   FIVE         DB      ?
   SIX          DW      257
;
   MODEL        ENDS
;
                END
```

Note that in the event of the initialised values being overriden, the new values must be listed inside the brackets <>. The fields are identified in the order left to right.

Thus, for COPY_3 four commas on the left are required to inform the assembler that 50 must be loaded into the field FIVE. On the other hand, field SIX, which keeps its value by default, is not shown in brackets.

The memory space is then as follows, in hexadecimal.

0 1	0 1	30 H
3 2	4 1	2E H
0 5	0 7	2C H
OFFSET	ADDRESS_1	2A H
0 F	F E	28 H
0 1	0 1	26 H
3 2	4 1	24 H
0 5	0 7	22 H
OFFSET	ADDRESS_1	20 H
0 F	F E	1E H
0 1	0 1	1C H
3 2	4 1	1A H
0 5	0 7	18 H
OFFSET	ADDRESS_1	16 H
0 F	F E	14 H <= COPY_3
0 0	F F	12 H
X X	4 1	10 H
0 5	0 7	0E H
0 0	0 0	0C H
0 F	F E	0A H <= COPY_2
0 1	0 1	08
X X	4 1	06
0 5	0 7	04
OFFSET	ADDRESS_1	02
0 F	F E	00 <= COPY_1

Example 3.12 shows which fields can have their initial values overridden.

Example 3.12

```
   OVERRIDE_?  STRUC
;  ---------

     DB    ?                            ; YES single fields, therefore overridable.
     DB    1,2,3                        ; NO multiple  fields, therefore  not over_
                                        ; ridable.
  T  DQ    10 DUP(?)                    ; NO
     DB    'MESSAGE'                    ; YES overridable by  another string of the
                                        ; same maximum length. If string is shorter
                                        ; it will be completed with  the last  let_
                                        ; ters of the word MESSAGE.
  M  DB    'MESSAGE_1 ','MESSAGE_2'     ; NO
     DD    OFFSET T                     ; YES
     DW    M                            ; YES
     DW    T,M                          ; NO
     DT    ?,2,3                        ; NO
;
   OVERRIDE_?  ENDS
;
               END
```

3.6.3 Reference to structured variables

It is possible to make reference to a particular field of a structured variable, in an instruction operand. To do this, the name of the structured variable is followed by a dot and the name of the field.

Example 3.13

```
               NAME    MODULE_A
;              ================

               N       EQU     3
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ====================================
   ADDRESS_1    LABEL   WORD
   COPY_1       MODEL   < >
   COPY_2       MODEL   <,0,,,,255>
   COPY_3       MODEL   3 DUP(<,,,,50>)
;
  DATA_TABLE  ENDS
;
   MODEL        STRUC
;  --------
   ONE          DW      OFFEH
   TWO          DW      OFFSET ADDRESS_1
   THREE        DB      7,5
   FOUR         DB      'A'
   FIVE         DB      ?
   SIX          DW      257
;
   MODEL        ENDS
```

```
;
   CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; ====================================
      ASSUME  CS:CODE_PP,DS:DATA_TABLE              ; ASSUME: see chapter 4

;        .
        MOV     AL,COPY_1.THREE          ; 07 stored in AL (OFFSET = 04)
        MOV     CL,COPY_1.THREE+1        ; 05 stored in CL (OFFSET = 05)
 ;       .
        MOV     DL,COPY_3.FIVE((N-1)*TYPE COPY_3)    ; (N-1)*TYPE COPY_3 = 2*10
                                         ; 50 stored in DL (OFFSET= 14H+7+20= 2FH)
 ;       .
        MOV     BX,OFFSET COPY_3         ; OFFSET COPY_3 = 14H + OFFSET COPY_1
                                         ; in the segment DATA_TABLE
 ;       .
        MOV     SI,(N-1)*TYPE COPY_3     ; 20 = 14H stored in SI
 ;       .
        MOV     AH,[BX][SI].FIVE         ; 50 stored in AH (OFFSET = 2FH)
;
   CODE_PP   ENDS
;
        END
```

4 Access to Data

4.1 Operands

Transfers are for the most part executed with the aid of a two-operand instruction; the left operand represents the destination, while the right operand represents the source.

The destination operand is a register or a memory location. The source operand can be a register, a memory location, or an immediate data item.

It should be noted that the two operands cannot both be memory locations. In other words, direct memory to memory transfer is not allowed. The MOVS instruction does in fact constitute an exception to this rule, but it is generally used for block transfers.

Finally, the type of the two operands is necessarily the same. This rule allows the assembler to determine the type of an operand, when it cannot know it, with the help of its correspondent.

The HIGH or LOW operators allow a memory word type to be modified, if this is necessary (see below).

For single operand transfer instructions, the second operand is implied as follows.

An implicit destination

- A register, usually the accumulator, for the operations of multiplication, division or repetitive transfer. For example, MUL BX; implies AX * BX -> AX and DX (32 bits).
- The top of the 8087 stack. For example, FST ST(1); implies ST(1) -> ST(0).
- JMP and CALL; imply that IP or CS and IP are the destination.
- Write operations to the stack. Example, PUSH BX.

An implicit source

- Read operations from the stack. For example, POP BX; implying that the word pointed to by SS:[SP] is the source and the register (BX in this case) is the destination.
- Iterative transfers which may include no operand (see the discussion of the instruction set in chapter 6).

J.-M. Trio, *8086–8088 Architecture and Programming*
© English Language edition, Macmillan Publishers Ltd 1985

4.1.1 Register type operands

The different possible register operands are described below.

Segment registers (16 bits)
CS, DS, SS, ES

Pointer and index registers (16 bits)
SP, BP, DI, SI

(These registers can also serve as general registers but only in 16 bits.)

General registers (16 bits)
AX, BX, CX, DX

Although capable of being used as required in all the instructions, each of these registers has a privileged use.

- Accumulator AX is used in preference for all numeric operations. Its use is obligatory for multiplication and division, as well as for communicating with any peripherals in the I/O field and certain repetitive operations.
- Base BX can be classified in the preceding category of pointers. When loaded with an offest address, it allows indirect addressing, associated as required with a variable name and an index.
- Register DX is associated with the accumulator to form the 32-bit word necessary for multiply and divide operations.

General registers (8 bits)
AH, AL, BH, BL, CH, CL, DH, DL

Each of the general 16-bit registers can be accessed as a pair of 8-bit registers as shown below.

15 8	7 0	
AH	AL	AX
BH	BL	BX
CH	CL	CX
DH	DL	DX

- AL necessarily looks after all 8-bit transfers with I/O peripherals which are not memory mapped.
- CL is used as the count for shift and bit rotation operations on memory words, 16 or 8 bit registers.

Numeric calculation registers within the 8087 (80 bits)
ST or ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7)

Organised as stacks, these registers always work in temporary real on 80 bits (see chapter 7).

Examples of the use of these register operands have been provided in previous chapters. Special use of these operands will be illustrated when we come to examine the instruction set and the following examples.

4.1.2 Immediate operands

Being located of course in the source, these operands are either numbers or the result of an evaluation by the assembler in 8 or 16 bits depending on the size of the destination operand.

As pure numbers are involved (that is, those not associated with a type), the size has to be specified to the assembler when this information is not provided by the destination operand. This is particualrly true of indirect addressing by base and index.

Example 4.1

```
   CODE  SEGMENT  BYTE  PUBLIC  'ROM'
; =================================
       ASSUME  CS:CODE
;
   PREPARATION  PROC
 ; ----------------
 ;             .
               MOV    AL,3                  ; 00000011B stored in AL
 ;             .
               MOV    BX,5                  ; 0005 H stored in BX
 ;             .
               MOV    BYTE PTR [BP],101B    ; Destination  type is unknown to
                                            ; Assembler and therefore must be
                                            ; indicated by using BYTE PTR.
 ;             .
;
   PREPARATION  ENDP
;
   CODE  ENDS
;
               END
```

4.1.3 Memory type operands

Strictly speaking, only registers are not memory (or peripheral) operands since the immediate operands are also located in memory, but in the very body of the instruction and because of this they present no problem of access to the assembler.

The same does not go for other types of operand called ´memory´ for which the assembler has continually to determine the following

- the segment value (see the directive ASSUME below)
- the type, already called (see operator PTR below)
- the offset

In this section we shall examine the offset more closely.

Jump and branch operands

For the most part, the jump or branch, whether NEAR or FAR, is direct and the operand provides either the address to load into IP alone, or the values to load into IP and CS.

However, it is possible to make jumps (JMP) and branches (CALL) indirectly. In this case, the operand provides the location of the branch address.

This location may be a register, a memory pointer name or a reference, itself indirect, with the help of the base or index register.

Example 4.2 illustrates simple and double indirect branches. It also shows the use of branch tables, which allow many programs to be simplified (equivalent to the Pascal instruction CASE OF).

Example 4.2

```
                NAME    MODULE_C
;               ================

        CODE    GROUP   CODE_PROCESS,CODE_MEASURE
;
   DIRECTORY   SEGMENT   WORD   PUBLIC   'ROM'
;  =====================================
    SHORT_BRANCH_TABLE         DW      OFFSET CODE:PROC_1
                               DW      OFFSET CODE:PROC_2
                               DW      OFFSET CODE:PROC_3

    LONG_BRANCH_TABLE          DD      PROC_A
                               DD      PROC_B

    POINTER_1                  DW      OFFSET CODE:MEASURE
    POINTER_2                  DD      CALCUL
;
   DIRECTORY   ENDS
```

```
  CODE_PROCESS  SEGMENT  BYTE  PUBLIC  'ROM'
; =========================================
        ASSUME  CS:CODE,DS:DIRECTORY             ; CODE_PROCESS segment is part of
                                                 ; the group CODE. CS contains the
                                                 ; value of CODE when jumping.
  PROCESSING  PROC
; ---------------
                MOV     AX,DIRECTORY             ; Optional if calling program has
                MOV     DS,AX                    ; already loaded DS. DS is set to
                                                 ; point at DIRECTORY as ASSUME has
                                                 ; indicated to Assembler.
;               .
                MOV     CX,10
   AGAIN:
;               .
;               .
                LOOPZ   AGAIN                    ; Direct jump as  long as CX  is
                                                 ; not equal to zero or a termina_
                                                 ; ting condition is not fulfilled.

                JCXZ    END_AGAIN                ; Direct jump if CX = 0.
;               .
;               .
  END_AGAIN:
;               .
;               .
                LEA     BX,AGAIN                 ; Offset  of  AGAIN,  within  the
                                                 ; group, is loaded in BX.
;               .
;               .
                JMP     BX                       ; Single indirect jump.
                                                 ; NOTICE THE ABSENCE OF BRACKETS.

;               .
                MOV     BX,OFFSET POINTER_1
                CALL    WORD  PTR [BX]           ; BX contains the address of  the
                                                 ; location in which is the offset
                                                 ; of the jump ( Double indirect )
                                                 ; NOTICE THE PRESENCE OF BRACKETS.
;               .
                MOV     BX,OFFSET POINTER_2
                CALL    DWORD  PTR [BX]          ; Double indirect by long pointer
                                                 ; FAR jump.
;               .
;               .
                MOV     AX,2                     ; AX=2
                MOV     DI,0                     ; Faster to use XOR DI,DI.
                MOV     SI,0                     ; XOR SI,SI is faster.
                ADD     SI,AX                    ; SI=2
                ADD     DI,AX                    ; DI=2
                CALL    SHORT_BRANCH_TABLE[DI]   ; Indirect jump to PROC_2
                                                 ; WARNING: increase DI by 2.
;               .
                ADD     SI,AX                    ; 04 stored in SI
                CALL    LONG_BRANCH_TABLE[SI]    ; Indirect jump at PROC_B
                                                 ; WARNING: INCREASE SI BY 4.
;               .
;
  PROCESSING  ENDP

  PROC_1  PROC                                   ; NEAR: CS=CODE unchanged.
; -----------
;               .
;               .
                RET
;
  PROC_1  ENDP
```

```
 PROC_2  PROC                                 ; NEAR: CS=CODE unchanged.
; -----------
;              .
;              .
               RET
;
 PROC_2  ENDP
;
  CODE_PROCESS  ENDS
;

 CODE_MEASURE  SEGMENT  BYTE  PUBLIC  'ROM'
; =========================================
       ASSUME  CS:CODE                        ; This  segment is part of  group
                                              ; CODE. CS=CODE remains unchanged
                                              ; during jumps.
 PROC_3  PROC                                 ; NEAR:CS=CODE remains unchanged.
; -----------
;              .
               RET
;
 PROC_3  ENDP

 MEASURE  PROC                                ; NEAR: CS=CODE remains unchanged.
; -----------
;              .
               RET
;
 MEASURE  ENDP
;
  CODE_MEASURE  ENDS

 CODE_CALCUL  SEGMENT  BYTE  PUBLIC  'ROM'
; =========================================
       ASSUME  CS:CODE_CALCUL                 ; This segment  IS  NOT  PART  OF
                                              ; group CODE. CS=CODE_CALCUL when
                                              ; jumps.
 CALCUL  PROC  FAR
; ---------------
;              .
               RET
;
 CALCUL  ENDP

 PROC_A  PROC  FAR
; ---------------
;              .
               RET
;
 PROC_A  ENDP

 PROC_B  PROC  FAR
; ---------------
;              .
               RET
;
 PROC_B  ENDP
;
  CODE_CALCUL  ENDS
;
               END
```

Data operands

Here again, apart from segment and type, the assembler must be able to find the offset of the memory location to be reached.

This access can be simple, indexed or structured.

Simple access

This consists in naming the variable in order to reach the value contained in this location.

Example 4.3

```
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; =======================================
   NUMBER     DB     5
   VALUE_1    DW     2007H
;
  DATA_TABLE        ENDS

  CODE  SEGMENT  BYTE  PUBLIC  'ROM'
; =================================
      ASSUME  DS:DATA_TABLE

              MOV     AX,DATA_TABLE
              MOV     DS,AX
;             .
              MOV     AL,NUMBER             ; 05H stored in AL
;             .
              MOV     DX,VALUE_1            ; 2007H stored in DX
;             .
              MOV     AL,BYTE PTR VALUE_1   ; 07H stored in AL (See PTR below)

  CODE  ENDS
;
              END
```

Indexed access

All or part of the offset value of the physical location to be reached is contained in a base register and/or a register index.

In addition, it is possible to add to the value of these registers a supplementary constant displacement.

Several different forms of writing are possible.

When this index value is added to a variable name, the assembler can determine the type and the segment (see ASSUME).

If this is not the case, this fact must be specified.

Example 4.4

```
  DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
; ======================================
   CONST_87    LABEL   WORD

   PARAM_A     DQ      8990000000000
   ADD_BFR     DD      BUFFERS

   TABLE_X_Y   LABEL   WORD

   X1          DW      11
   X2          DW      12
   X3          DW      13
   Y1          DW      24
   Y2          DW      26
   Y3          DW      32
;
  DIRECTORY  ENDS

  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ======================================
   BUFFERS     LABEL   WORD

   BUFFER_1    DT      ?
   BUFFER_2    DT      ?
;
  DATA_TABLE  ENDS

  CODE_PROCESS  SEGMENT  BYTE  PUBLIC  'ROM'
; ======================================
        ASSUME  CS:CODE_PROCESS,DS:DIRECTORY    ; Starting from this line, DS  is
                                                ; assumed to  contain  the  value
                                                ; DIRECTORY. If  calling  program
                                                ; has not loaded DS, called  pro_
                                                ; gram must do the loading.
 PROCESSING  PROC
; ---------------
;               .
                FILD    PARAM_A                 ; The integer value 8990000000000
;               .                                 is loaded into ST(0).
                LES     BX,ADD_BFR              ; DATA_TABLE  value is loaded in_
                                                ; to ES. OFFSET of BUFFERS within
                                                ; DATA_TABLE is loaded into BX.
                MOV     SI,TYPE BUFFER_1        ; 10 stored in SI
                FSTP    TBYTE PTR ES:[BX]       ; Temporary real number contained
                                                ; within ST(0) is stored into BUF_
                                                ; FER_1. (8990000000000 in binary
                                                ; notation  with  80  bits).  See
;               .                                 chapter 7.
                FLD     TBYTE PTR ES:[BX]       ; BUFFER_1 stored in ST(0)
                FSTP    TBYTE PTR ES:[BX][SI]   ; ST(0) stored in BUFFER_2
;               .


                MOV     BX,OFFSET TABLE_X_Y
                MOV     AX,DIRECTORY
                MOV     ES,AX
                MOV     SI,TYPE X1              ; 2 stored in SI
                MOV     DI,Y1-X1                : 6 stored in DI
                MOV     AX,ES:TABLE_X_Y[SI]     ; 12 stored in AX. The name
                                                ; TABLE_X_Y gives the type.
                ADD     AX,ES:TABLE_X_Y[DI]     ; X2+Y1 = 12+24 stored in AX
;               .
```

```
              MOV     AX,ES:[BX]             ; X1=11 stored in AX.Type unknown
                                             ; to  Assembler. It  must be  the
                                             ; same of the type of AX  (WORD).
;             .
              MOV     DX,ES:[BX][SI]         ; X2=12 stored in DX
              MOV     CX,ES:[BX][SI]+2       ; X3=13 stored in CX
              MOV     CX,ES:[BX+4][SI-2]     ; Identical: 13 stored in CX
              MOV     CX,ES:[BX+SI+2]        ; Identical: 13 stored in CX
              MOV     CX,ES:TABLE_X_Y[SI+2]  ; Identical: 13 stored in CX
;             .
;
  PROCESSING  ENDP
;
  CODE_PROCESS  ENDS

;
              END
```

Access to structured variables

Apart from the above-mentioned possibilities we can also use structured variable field names.

This leads not only to a change of the offset, as in the case of a constant displacement, but in particular to a change of type to that of the field.

Example 4.5

```
              NAME    MODULE_A
;             ================

   S  STRUC
;  - - - - -
   A          DB      0
   B          DW      0
   C          DD      ?
   D          DW      0
   E          DB      ?
;
   S  ENDS
;
  DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
; =====================================
   FILE_1     S       <1,2,FILE_2,,0>
;
  DIRECTORY  ENDS

  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ======================================
   FILE_2     S       <,,1.23,,0>
;
  DATA_TABLE  ENDS


  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; ==================================
       ASSUME  DS:DIRECTORY
;             .
;             .
              MOV     AX,DIRECTORY
              MOV     DS,AX
```

```
            LES     BX,FILE_1.C                ; TYPE of C = DWORD (Pointer)
                                               ; SEGMENT of FILE_2 stored in ES
                                               ; OFFSET of FILE_2 stored in BX
            MOV     AL,ES:[BX].E               ; TYPE of E = BYTE
                                               ; OFFSET = OFFSET of FILE_2 +9
                                               ; 0 stored in AL
;           .
            FLD     ES:[BX].C                  ; TYPE of C = DWORD (Real Simple)
                                               ; 1.23 stored in ST(0).  Warning:
                                               ; MICROSOFT assembler  does  not
                                               ; follow the rule of IEEE numeric
                                               ; representation (See DD in chap_
                                               ; ter 3).
;           .
            MOV     AX,FILE_1.B                ; 0002 stored in AX
            MOV     ES:[BX].D,AX               ; 0002 stored in FILE_2.D
                                               ; (initial value)
;           .
;
  CODE_PP   ENDS
;
            END
```

4.2 Attribute Operators

These operators allow operand attributes to be modified, the value of one of these attributes to be used, or the characteristics of a RECORD to be used.

Several of these operators are used frequently. They have already been employed in the previous examples.

4.2.1 Attribute corrector operators

PTR operator

This needs to be used whenever one operand type does not correspond to the type of another operand, or when the assembler cannot determine it.

Example 4.6

```
            NAME    MODULE_A
;           ===============

      CODE  GROUP   CODE_PP,CODE_PREP
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ========================================
   RESERVED   DW     1234H
   JUMP       DW     OFFSET ADDRESS_1
              DW     OFFSET CODE:ADDRESS_2
              DW     CODE
   VALUES     DD     ?
              DQ     5E10                 ; WARNING :  MICROSOFT  assembler
                                          ; does not follow the IEE rule of
                                          ; representation (See chapter 3).
              DT     ?
;
  DATA_TABLE  ENDS

  CODE_PP   SEGMENT  WORD  PUBLIC  'ROM'
; ==================================
       ASSUME  DS:DATA_TABLE,CS:CODE
;           .
            MOV     AX,DATA_TABLE
            MOV     DS,AX
;           .
   ADDRESS_1: MOV   AL,BYTE PTR RESERVED     ; 34H stored in AL
            MOV     CL,BYTE PTR RESERVED+1   ; 12H stored in CL
;           .
```

```
             MOV    AX,RESERVED            ; 1234H stored in AX
             MOV    WORD PTR VALUES,AX
             MOV    WORD PTR VALUES+2,0    ; VALUES = 00001234H
;            .
             MOV    BX,OFFSET VALUES       ; See also instruction LEA
             FILD   DWORD PTR [BX]         ; Loading of a long integer in
                                           ; ST(0) of 8087.
             FSTP   TBYTE PTR [BX]+12      ; Storage of ST(0) with temporary
                                           ; real format.
             FLD    QWORD PTR [BX]+4       ; Loading of a double precision
                                           ; real in ST(0) of 8087.
             MOV    BX,OFFSET JUMP         ; See also instruction LEA.
;            .
             JMP    WORD PTR [BX]          ; IP= OFFSET of ADDRESS_1 within
                                           ; CODE_PP.
;            .
             CALL   DWORD PTR [BX]+2       ; IP= OFFSET of ADDRESS_2 within
                                           ; CODE. CODE within CS.
;            .
  CODE_PP   ENDS
;
  CODE_PREP  SEGMENT  BYTE  PUBLIC  'ROM'
; ========================================
        ASSUME  CS:CODE
;            .
 PREPARATION  PROC  NEAR
; ---------------------
;            .
  ADDRESS_2:
;            .
             RET
;
 PREPARATION  ENDP
;
  CODE_PREP  ENDS
;
             END
```

In example 4.6 ´hand calculation´ of the values +12 and +4 in instructions FSTP and FLD must be avoided, as this could lead to errors. Variable field modifications are difficult and the readability is very poor. Use of a structure would have avoided these advantages.

Additionally, types would have been provided by the field types of the structure, thus avoiding use of PTR in structions FILD, FSTP and FLD (see example 4.5).

Segment prefix

The operator is the colon ´:´, preceded by the name of a segment register, by the name of a segment - either nominally or by means of the SEG operator (see section 4.2.2), or by the name of a group.

In the last-named case, the OFFSET attribute is recalculated to give the offset from the group base.

The use of these prefixes has already been shown in several examples.

SHORT operator
This is used in the operand of a JMP or a CALL, and it allows a byte of code to be saved in the instruction coding and thus speeds up execution. In order to be able to make use of it, the address to be reached must lie between -127 and +126 bytes of the JMP or CALL instruction.

This operator is used implicitly by the assembler if the branch address at assembly time precedes the transfer instruction (see example 4.8).

HIGH and LOW operators
These allow a byte in a 16-bit memory word or an immediate value to be isolated.

Example 4.7

```
  CODE   SEGMENT   BYTE   PUBLIC   'ROM'
; ==================================
       ASSUME  CS:CODE
;              .
;              .
   TRIAL       DW      1234H
   HALF        DW      ?
;              .
               MOV     AL,LOW TRIAL            ; 34H stored in AL
;              .
               MOV     AL,BYTE PTR TRIAL       ; 34H stored in AL (identical)
;              .
               MOV     AL,HIGH TRIAL
               MOV     AH,LOW TRIAL            ; 3412H stored in AX
               MOV     DX,0
               MOV     DH,LOW TRIAL            ; 34H stored in DH
               MOV     BYTE PTR HALF,DH
               MOV     BYTE PTR HALF+1,DL      ; 0034H stored in HALF
;              .
  CODE   ENDS
;
               END
```

THIS operator
THIS allows a name and a type to be given to the current value of the location counter. When used in the EQU directive, it is of little value since it causes double use with the LABEL directive for the definition of variables and the $ symbol for determining the length of jumps in transfer instructions.

4.2.2 Attribute use operators

These operators permit access to the values attributed by the assembler to each variable or label.

SEG provides the segment value of the variables or label. It is very useful when assigned to the variables or labels defined in other modules and passed by EXTRN (see below). In reality, in such a case, the segment name in which these symbols have been defined is not always known to the programmer.

OFFSET provides the offset value, calculated at the time of LINK.

TYPE assigns to each variable type the number of bytes used by that type or the attribute NEAR or FAR for labels.

LENGTH provides the number of units in a variable. In fact, it resupplies the DUP number used to define the variable.

SIZE provides the byte number of a variable:
SIZE = LENGTH * TYPE

Example 4.8

```
;       This example demonstrates some of the flexibility and clarity resulting
;from the systematic use of operators described above.
;
;       The procedure SUM is used for totalling the 150 items, in real simple
;precision, of any of the 20 vectors of TABLE_SHORT. The same procedure SUM
;then totals the 100 items, in real double precision, of any of the 20 vectors of
;TABLE_LONG.


                NAME    MODULE_A
;               ================

   REAL_SHORT  STRUC
;  ----------
   DATA_SHORT           DD              150 DUP(?)
   SUM_SHORT            DD              ?
;
   REAL_SHORT  ENDS

   REAL_LONG  STRUC
;  ----------
   DATA_LONG            DQ              100 DUP(?)
   SUM_LONG             DQ              ?
;
   REAL_LONG  ENDS
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ========================================
   TABLE                LABEL           WORD

   TABLE_SHORT          REAL_SHORT      20 DUP(<>)
   TABLE_LONG           REAL_LONG       20 DUP(<>)
;
  DATA_TABLE  ENDS

  CODE  SEGMENT  BYTE  PUBLIC  'ROM'
; ==================================
      ASSUME  CS:CODE,ES:SEG TABLE
;               .
                MOV     AX,SEG TABLE
                MOV     ES,AX
;               .
;               .
```

```
                LEA     BX,TABLE_SHORT           ; LEA : Load  Effective  Address.
                                                 ; BX  =  OFFSET  of  TABLE_SHORT
                                                 ; stored in DATA_TABLE.
                MOV     SI,TYPE DATA_SHORT       ; SI = 4
                MOV     DI,TYPE REAL_SHORT       ; DI = 604 (150*4+4)
                MOV     CX,LENGTH DATA_SHORT     ; CX = 150
                MOV     AX,3
                CALL    SUM                      ; Determines the sum of the third
                                                 ; vector of TABLE_SHORT.
;               .
                ADD     BX,SIZE TABLE_SHORT      ; BX = BX+20*604
                MOV     SI,TYPE DATA_LONG        ; SI = 8
                MOV     DI,TYPE REAL_LONG        ; DI = 808
                MOV     CX,LENGTH DATA_LONG      ; CX = 100
                MOV     AX,14
                CALL    SUM                      ; Sum of the  fourteenth  vector.
;               .

 SUM  PROC
; ---------
                DEC     AX                       ; AX = AX-1
                MUL     DI                       ; AX = AX*DI
                ADD     BX,AX                    ; BX points  the start of the nth
                                                 ; vector.
                MOV     AX,SI
                XOR     SI,SI                    ; SI=0 initialisation of the index
                FLDZ                             ; ST(0) = 0 initialisation of the
                                                 ; sum.
   AGAIN:       CMP     AX,4                     ; Comparison  of  SI  with  4.
                JNE     LONG                     ; Jump of the case of real  long.
                FLD     DWORD PTR ES:[BX][SI]    ; Loading of the simple precision
                                                 ; item of the vector.ST(0) within
                                                 ; ST(1) and element within ST(0).
                JMP     SHORT NEXT               ; Short jump.
   LONG:        FLD     QWORD PTR ES:[BX][SI]    ; Loading of the double precision
                                                 ; item of the vector.

   NEXT:        FADDP   ST(1),ST(0)              ; ST(1) = ST(1) + ST(0)
                                                 ; then ST(0) = ST(1)
                ADD     SI,AX                    ; SI= SI+AX increase of the index
                LOOP    AGAIN                    ; CX = CX-1
                                                 ; Looping if CX is not equal to 0
                CMP     AX,4
                JNE     OUTPUT
                FSTP    DWORD PTR ES:[BX][SI]    ; Outcome as  real short in  SUM_
                                                 ; SHORT
                JMP     SHORT EXIT
   OUTPUT:      FSTP    QWORD PTR ES:[BX][SI]    ; Outcome as  real  long in  SUM_
                                                 ; LONG
   EXIT:        RET
;
 SUM      ENDP
;
   CODE   ENDS
                        END
```

4.2.3 Operators specific to RECORD

There are three operators which are specific to record fields:

1. The right shift number required for a byte or a word field. The operator is the field name itself.

2. MASK, which allows a particular field to be isolated.
3. WIDTH, which provides the number of bits in a field or in the RECORD itself. (See also chapter 3.)

4.2.4 Arithmetic operators

These operators apply to absolute numbers and give results that are also absolute numbers.

Their operations are carried out during assembly and they should not be confused with instructions that can be executed using the same mnemonics (see example 4.9).

The format of the operands and results is thus that of the assembler, that is, 16 bits (signed or unsigned).

HIGH and LOW. These serve to isolate the high order and low order bytes of an expression.

```
HIGH(1234H)  has the value 12H
LOW(1234H)   has the value 34H
```

*, / and MOD. 20 * 4 has the value 50H. Care should be taken to avoid obtaining results that are too large as they could affect the 16th bit and thus modify the interpretation of the sign for numerical values.

```
1234H/256       has the value 12H
1234h MOD 256   has the value 34H
```

SHL and SHR. These operators carry out bit shifts in expressions, either to the left (SHL) or to the right (SHR). The freed bits are reset.

```
11H SHL 3  has the value 88H
11H SHR 2  has the value 4
```

+ and -. These have their normal function, and their use has already been illustrated in the previous examples.

4.2.5 Boolean operators

These also serve to allow the assembler to evaluate expressions. The result is either all bits at 1 if the proposition is true, or all bits at 0 if it is false.

There are the following six operations:

```
EQ : equal                  ; 3 EQ 11B = 0FFFFH (true)
NE : not equal              ; 3 NE 11B = 0000 (false)
LT : less than              ; 3 LT 01B = 0000 (false)
LE : less than/equal to     ; 3LE 30 MOD 4 = 0000 (false)
GT : greater than
GE : greater than/equal to
```

4.2.6 Binary operators

NOT ; provides the one´s complement
NOT 32H = NOT (00100000B) = 11011111B = 0DFH
AND ; 1AH AND 0FH = 00011010B AND 00001111B = 1010B = 0AH
OR ; 1AH OR 0FH = 1FH
XOR ; 1AH XOR 0FH = 00010101B = 15H

4.2.7 Operator hierarchy

It is useful to know the operator priority order so that one can insert brackets if necessary to ensure that operations are carried out in the required sequence.

The hierarchy is as follows, in decreasing order of priority:

*1 : Brackets, <> for RECORD, square brackets for structures, the operator ´.´ of a structure, LENGTH, SIZE, WIDTH and MASK.
*2 : PTR, OFFSET, SEG, TYPE, THIS and segment prefix.
*3 : HIGH, LOW.
*4 : *, /, MOD, SHL, SHR.
*5 : +, -.
*6 : EQ, NE, LT, LE, GT, GE.
*7 : NOT.
*8 : AND.
*9 : OR, XOR.
*10 : SHORT.

4.2.8 EQU directive

The EQU directive, already described in part, has two very different sorts of use.

Evaluation of an expression

The result is a pure number and EQU allows it to be named.

This feature is extremely practical since it allows the constants used in many instructions to be put at the front of a program. If a value has to be modified, only one line has to be corrected. In addition, program readability is greatly improved.

Replacing a string of characters

In this case, no evaluation is carried out. There is simply a substitution of one character string by another.

This facility makes it possible to reduce writing considerably or create one´s own mnemonics. Even instruction mnemonics can be changed (but not both at once within one and the same EQU).

Example 4.9

```
               NAME    MODULE_C
;              ===============

   BAUDOT      RECORD  A:3,B:7,C:5

   B333        EQU     BAUDOT<3,3,3>              ; B333 = 0110000011000011B = 3063H
   B505        EQU     BAUDOT<5,0,5>              ; B505 = 1010000000000101B = 5005H
   COMBINE     EQU     B333 OR B505               ; COMBINE=1110000011000111B= 7067H
   LOAD        EQU     MOV
   GRID1       EQU     0101010101010101B
   GRID2       EQU     1010101010101010B
   CHOICE      EQU     2
   P           EQU     CS:[BX]
;
   CODE  SEGMENT  BYTE  PUBLIC  'ROM'
;  ================================
       ASSUME  CS:CODE
;              .
   TABLE       DW      1AH
               DW      OFFH
;              .
               LEA     BX,TABLE
               XOR     SI,SI                      ; SI = 0 : shows the fastest  way
                                                  ; of initiating a register at 0
               LOAD    AX,P[SI]                   ; Equivalent to MOV AX,CS:[BX][SI]
                                                  ; AX = 1AH
               AND     AX,B333 AND GRID1          ; Assembler  evaluates  B333  AND
                                                  ; GRID1  and  places  the  result
                                                  ; 1041H as immediat value in ope_
                                                  ; rating code. Instruction AND,at
                                                  ; run time, will lead CPU to ope_
                                                  ; ration  1AH AND 1041H=0  stored
                                                  ; in AX
;              .
               INC     SI                         ; Increment : SI = SI + 2.
               INC     SI                         ; Must be prefered to instruction
                                                  ; ADD SI,2  which is longer.
               LOAD    AX,P[SI]                   ; AX = OFFH

       XOR    AX,B333 AND ((GRID1 AND CHOICE EQ 1) OR (GRID2 AND CHOICE EQ 2))
                                                  ; The assembler  evaluates  first
                                                  ; the right  operand of the  ins_
                                                  ; truction. CHOICE EQ 1 is  false
                                                  ; therefore equal to  zero. GRID1
                                                  ; AND CHOICE EQ 1 is equal to zero
                                                  ; However  CHOICE  EQ 2  is  true
                                                  ; therefore equal to  OFFFFH  and
                                                  ; GRID2  is  selected.  B333  AND
                                                  ; GRID2 = 2022H   is   loaded  as
                                                  ; immediate value in the instruc_
                                                  ; tion.  At  run  time,  OFFH XOR
                                                  ; 2022H = ODDH stored in AX.
;              .
;              .
;
   CODE  ENDS
                         END
```

4.3 The ASSUME Directive

Often in the previous chapters the following problem has arisen for the 16-bit assembler: the memory field is segmented and there are many segment and offset combinations that allow a particular absolute value to be reached. Now, on execution the segment value is necessarily provided by one of the four segment registers.

From then on, the logical route for the assembler when it needs to reach a variable or label location is by establishing the following:

- in which segment or group is this variable or label located?
- which is the register loaded with this segment or group value?

Thanks to its symbol arrays, the assembler can easily solve the first problem, more easily even that the programmer, since the latter can ask it, by means of the SEG directive, to find the name and thus the value of the segment in which a particular variable or label is located.

On the other hand, the second question poses, a priori, an insuperable problem. How can the assembler know the value that will be held in the CPU segment registers at the time of execution of the instruction that it is in the process of coding?

Similarly, how could it chose one register in preference to another or how can it tell the programmer that none of the register contents is suitable?

It would have to carry out an execution simulation in order to establish how the programmer loaded such and such a register.

And even then, what happens if this loading has been carried out in another assembly module?

It is therefore essential for the programmer to indicate to the assembler the contents of segment registers, such as he has foreseen them to be at the time of execution, and he will pay the penalty if he does not load them correctly.

Against this, the assembler automatically generates corrective prefixes for segment use if the implicit segment is not the one required. It also corrects the offsets if the segment registers are loaded with group bases.

Furthermore, the assembler is then able to indicate to the programmer if any segment register does not allow correct access.

The ASSUME directive provides this information link between the programmer and the assembler.

4.3.1 Access in the same module

This is the simplest case. As soon as the assembler knows the ASSUME directive, it has all the necessary information.

Example 4.10

```
              NAME    MODULE_C
;             ===============

  ADD_INT  SEGMENT  WORD  AT  0
; ==============================
              ORG      00
   DIVISION_0 DD       ?
              ORG      08
   NMI        DD       ?
;
  ADD_INT  ENDS

  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ========================================
   TABLE       DW      100 DUP(?)
   RESULT      DQ      ?
;
  DATA_TABLE  ENDS

  DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
; =======================================
   INTER_0     DD      EXCEPTION_0
   INTER_2     DD      NON_MASKABLE
   ADD_VAR_1   DD      VAR_1
;
  DIRECTORY  ENDS

  CODE_INT  SEGMENT  BYTE  PUBLIC  'ROM'
; ======================================
 EXCEPTION_0  PROC  FAR
; --------------------
;             .
;             .
              IRET
;
 EXCEPTION_0  ENDP

 NON_MASKABLE  PROC  FAR
; ----------------------
;              .
;              .
               IRET
;
 NON_MASKABLE  ENDP
;
  CODE_INT  ENDS
  ;
 CODE   SEGMENT   BYTE   PUBLIC   'ROM'
; ==============================
        ASSUME  CS:CODE

  VAR_1         DB      5AH
;               .
;               .
                MOV     AX,DATA_TABLE
                MOV     DS,AX
;               .
                MOV     TABLE+6,03H             ; FAULT: assembler does not  know
                                                ; that DS, implicitly  used, con_
                                                ; tains  DATA_TABLE. However  the
                                                ; execution   would   have   been
                                                ; correct.
        ASSUME  DS:DATA_TABLE
;               .
                MOV     TABLE+6,03H             ; CORRECT.
;               .
                MOV     WORD PTR RESULT,01F5H   ; CORRECT:  RESULT  belongs   to
                                                ; DATA_TABLE,  contained   in  DS
                                                ; which is implicitly used.
;               .
                LES     BX,ADD_VAR_1            ; FAULT: no segment  register  of
                                                ; CPU  contains  DIRECTORY. Error
                                                ; will be issued.
;               .
                MOV     AX,DIRECTORY
                MOV     ES,AX
;               .
;               .
        ASSUME  ES:DIRECTORY
;               .
                LES     BX,ADD_VAR_1            ; CORRECT:  Assembler   generates
                                                ; correcting prefix  in order  to
                                                ; utilize ES instead of DS.
        ASSUME  ES:SEG VAR_1

                MOV     AX,[BX]                 ; FAULT: Assembler cannot identi_
                                                ; fy VAR_1. [BX] is anonymous and
                                                ; works implicity with DS.
                                                ; NO ERROR IS ISSUED.

                MOV     AX,ES:[BX]              ; CORRECT.

                MOV     ES:TABLE,AX             ; FAULT:   Assembler   takes   ES
                                                ; register without any  checking.
                                                ; NO ERROR IS ISSUED.

                MOV     TABLE,AX                ; CORRECT.

                LEA     BP,TABLE                ; CORRECT.

                MOV     WORD PTR [BP+2],22H     ; FAULT: BP works implicitly with
                                                ; SS.
                                                ; NO ERROR IS ISSUED.
                MOV     BP,04
                MOV     TABLE[BP],44H           ; CORRECT: Assembler  can  verify
                                                ; access with DS implicitly used.
                                                ; The name has priority over  the
                                                ; base or the index.
        ASSUME  ES:DIRECTORY

                MOV     AX,WORD PTR INTER_0      ; FAULT: Assembler finds  INTER_0
                                                 ; in DIRECTORY and generates  the
                                                 ; correcting prefix for using  ES
                                                 ; as indicated in ASSUME. However
                                                 ; ES  contains  the  segment  of
                                                 ; VAR_1. Assembler cannot replace
                                                 ; nor verify the right loading of
                                                 ; ES.
                                                 ; NO ERROR IS ISSUED.
                MOV     AX,DIRECTORY
                MOV     ES,AX
                MOV     DX,WORD PTR INTER_0      ; CORRECT: ES is used.

                MOV     AX,SEG DIVISION_0
                MOV     DS,AX

        ASSUME  DS:SEG DIVISION_0

                MOV     WORD PTR DIVISION_0,DX   ; CORRECT: implied DS is checked.

                MOV     DX,WORD PTR INTER_0+2    ; CORRECT: ES is used.
                MOV     WORD PTR DIVISION_0+2,DX; CORRECT.
;               .
;               .
   CODE   ENDS
;
                END
```

Example 4.10 shows that the ASSUME directive is only taken into account for the instructions that follow it. It remains effective until the next ASSUME directive that bears on the same segment register.

In order to remove as ASSUME directive without defining a fresh one, the keyword NOTHING can be used.

Thus ASSUME DS:NOTHING cancels the effect of the last ASSUME directive bearing on DS.

ASSUME NOTHING frees all segment registers.

Example 4.10 also illustrates the difficulties encountered by the programmer when he works in a context that comprises several segments.

The following section outlines the simplifications brought through the use of the GROUP directive.

4.3.2 Access within a group

Example 4.11

```
                NAME    MODULE_C
;               ===============

   PUBLIC       NMI,TABLE,RESUL_W,INTER_2,ADD_VAR_1,LONG_TABLE
   EXTRN        NON_MASKABLE:FAR

   CODE         GROUP   CODE_PP                  ; See intermodule access.
   DATA         GROUP   DATA_TABLE,DIRECTORY
;
```

```
  ADD_INT   SEGMENT   WORD   AT   0
; ============================
               ORG      08
   NMI         DD       ?
;
  ADD_INT   ENDS

  DATA_TABLE   SEGMENT   WORD   PUBLIC   'RAM'
; ======================================
   TABLE       DW       100 DUP(?)
   RESUL_W     LABEL    WORD
   RESULT      DQ       ?
;
  DATA_TABLE     ENDS
;
   LONG_TABLE  EQU      LENGTH TABLE
;
  DIRECTORY   SEGMENT   WORD   PUBLIC   'ROM'
; =====================================
   INTER_2     DD       NON_MASKABLE
   ADD_VAR_1   DD       VAR_1
;
  DIRECTORY   ENDS
;

  CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; ===================================
      ASSUME   CS:CODE,DS:DATA,ES:DATA

   VAR_1       DW       5AH
;              .
   ENTRY_POINT:
;              .
               MOV      AX,DATA
               MOV      DS,AX
               MOV      ES,AX

               MOV      TABLE+6,03H              ; CORRECT: Assembler will  itself
                                                 ; add to 06 the  offset  existing
                                                 ; between DATA and DATA_TABLE

               MOV      WORD PTR RESULT,01F5H    ; CORRECT.

               MOV      RESUL_W,01F5H            ; Equivalent  to  preceding  ins_
                                                 ; truction.

               LEA      BX,ADD_VAR_1             ; EXACT: Assembler knows  through
                                                 ; Assume that the only possibili_
                                                 ; ty to reach ADD_VAR_1 is to use
                                                 ; the   group   DATA.   Therefore
                                                 ; the offset of ADD_VAR_1 is com_
                                                 ; puted then  placed as immediate
                                                 ; value in instruction with refe_
                                                 ; rence to DATA as base.

               MOV      AX,[BX]                  ; CORRECT: the offset of VAR_1 in
                                                 ; CODE_PP is loaded in AX.
      ASSUME  ES:NOTHING

               LES      BX,ADD_VAR_1             ; The  content  of  ADD_VAR_1  is
                                                 ; used here.  ES is  loaded  with
                                                 ; CODE_PP which is the segment of
                                                 ; VAR_1.  BX is  loaded  with  the
                                                 ; offset of VAR_1 in CODE_PP.
```

```
                MOV     AX,VAR_1                ; CORRECT: segment CS is issued.

                MOV     TABLE,AX                ; CORRECT.

                MOV     BP,OFFSET TABLE

                MOV     AX,[BX]                 ; FAULT: the offset contained  in
                                                ; BX is without reference to  the
                                                ; content  of DS implicitly used.
                                                ; NO ERROR IS ISSUED.

                MOV     WORD PTR [BP],11H       ; FAULT: BP works implicitly with
                                                ; SS.
                                                ; NO ERROR IS ISSUED.

                MOV     AX,ES:[BX]              ; CORRECT.

                MOV     WORD PTR DS:[BP],11H    ; FAULT: BP has been loaded  with
                                                ; offset  in  the  segment  DATA_
                                                ; TABLE.  The   offset   existing
                                                ; between  DATA  and  DATA_TABLE
                                                ; cannot  be  taken into  account
                                                ; by the Assembler.
                                                ; NO ERROR IS ISSUED.

                LEA     BP,TABLE                ; BP is loaded with the offset of
                                                ; TABLE within the group  because
                                                ; the directive ASSUME shows that
                                                ; there is no other access.

                MOV     WORD PTR DS:[BP],11H    ; CORRECT.

                MOV     BP,04
                MOV     TABLE[BP],44H           ; CORRECT.

                MOV     AX,WORD PTR INTER_2     ; CORRECT: DS is used.

                MOV     WORD PTR NMI,AX         ; FAULT: SEG_INT does not  belong
                                                ; to the Group.
                MOV     DX,ADD_INT
                MOV     ES,DX
                MOV     ES:WORD PTR NMI,AX      ; CORRECT.

        ASSUME  ES:SEG NMI

                MOV     AX,INTER_2+2
                MOV     WORD PTR NMI+2,AX       ; CORRECT.
;
   CODE_PP  ENDS
;
                END
```

4.3.3 Intermodular access

The PUBLIC directive, which is usually placed at the head of a module, indicates to the assembler that one or more variables or labels held in this module are used in other modules.

The assembler then places the attributes of these variables or labels in a table that will be exploited by LINK.

Additionally, when a module refers to an external variable or label it must place it in one or more EXTRN directives.

However, the assembler cannot wait for the LINK in order to resolve its immediate problems, that is, whether segment, type, offset within/outside a group.

For the type, the indication is carried in the EXTRN directive itself, after a colon ´:´ after the name. BYTE, WORD, QWORD, NEAR, FAR are allowed, as well as ABS for an absolute number defined using an EQU.

Note that QWORD, TBYTE, the names of structures and records are only supported in recent versions of the assembler.

For the segment, it is necessary to proceed to a reopening of the segment in the module, provided that this segment is known and that it includes a PUBLIC alignment attribute.

In this case, the EXTRN is placed in the segment.

If this is not the case, the SEG operation also allows the segment to be provided for the assembler, but the EXTRN directive must be placed outside any segment and it can no longer belong to a group.

Taking MODULE_C to be written as in example 4.11, example 4.12 shows the use of the EXTRN and PUBLIC directives.

Example 4.12

```
               NAME    MODULE_A
;              ================

               PUBLIC  PREPARATION

               EXTRN   NMI:DWORD,LONG_TABLE:ABS

   CODE        GROUP   CODE_PP,CODE_PREP
   DATA        GROUP   DATA_TABLE,DIRECTORY
;
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ======================================
               EXTRN   TABLE:WORD,RESUL_W:WORD
;
  DATA_TABLE  ENDS

  DIRECTORY  SEGMENT  WORD  PUBLIC  'ROM'
; =====================================
               EXTRN   INTER_2:DWORD,ADD_VAR_1:DWORD
;
  DIRECTORY  ENDS

  CODE_PP  SEGMENT  BYTE  PUBLIC  'ROM'
; ==================================
               EXTRN   VAR_1:WORD
;
  CODE_PP  ENDS
  ;
  CODE_PREP  SEGMENT  BYTE  PUBLIC  'ROM'
; ==================================
      ASSUME  CS:CODE,DS:DATA,ES:SEG NMI
;
 PREPARATION  PROC
; ----------------
            MOV     AX,DATA              ; DS loaded with  DATA in  accor_
            MOV     DS,AX                ; dance with ASSUME.

            MOV     AX,SEG NMI           ; ES loaded with  SEG NMI in  ac_
            MOV     ES,AX                ; cordance with ASSUME.

            MOV     AX,WORD PTR INTER_2  ; DS is used.
            MOV     WORD PTR NMI,AX      ; ES is used.

            MOV     AX,WORD PTR INTER_2+2
            MOV     WORD PTR NMI+2,AX

            MOV     DX,VAR_1             ; CORRECT: CS  is  used  and  the
                                         ; offset is  computed with  refe_
                                         ; rence to CODE.
            LEA     DI,VAR_1             ; Loading of offset with referen_
                                         ; ce to CODE, since there is no o_
                                         ; ther possibility through ASSUME
            MOV     DX,CS:[DI]           ; CORRECT.
;           .
;           .
            MOV     CX,LONG_TABLE
            XOR     DI,DI
   AGAIN:   MOV     TABLE[DI],0
            INC     DI
            INC     DI
            LOOP    AGAIN
;           .
;           .
            FLD     QWORD PTR RESUL_W    ; Access to double  real  outside
                                         ; of the module.
            RET
;           .
;           .
 PREPARATION  ENDP
;
  CODE_PREP  ENDS
;
            END
```

4.4 Use of the Stack

The stack is used primarily in assembler language programs for the temporary storage of variables, using the PUSH and POP instructions.

However, such usage brings with it the constraint of sorting the data into a very precise order - the inverse order of the items in the stack.

How then does one proceed to access a value that was not the last value pushed on to the stack?

Also, how does one choose a type other than WORD, which is enforced by the PUSH and POP instructions?

Furthermore, defining variables in data segments can seem clumsy when it is only a question of purely local variables that are of no importance for other procedures.

It is therefore necessary to be able to to work with direct access to the stack. For this, the systematic use of STRUCTURES is practically indispensable.

4.4.1 Local variables

The base register BP, associated implicitly with SS, is intended specifically for this use.

On the other hand, use of a structure allows both memory slots in the stack to be named, even though dynamic allocation is involved, and a type to be attributed to them.

Example 4.13

```
                NAME    MODULE_A
;               ================

                PUBLIC  PRODUCTS_XY

  TEST_X        EQU     2
  TEST_Y        EQU     3
;
 VAR_LOC  STRUC
; - - - - - - - -

  X             DW      ?
  X_REAL        DD      ?
  BUFFER        DQ      10 DUP(?)
  Y             DW      ?
  Y_REAL        DD      ?
  FORMER_BP     DW      ?
;
 VAR_LOC  ENDS
;
  VL            EQU     [BP - OFFSET FORMER_BP]
;
  CODE_PREP  SEGMENT  BYTE  PUBLIC  'ROM'
; ======================================
        ASSUME  CS:CODE_PREP
;
 PRODUCTS_XY  PROC  FAR
; ====================
                PUSH    BP                      ; Storage of register's actual
                                                ; value, in order not to prevent
                                                ; calling program from working
                                                ; with its own local variables.
                MOV     BP,SP                   ; BP points to the memory word
                                                ; containing FORMER_BP.
                SUB     SP,OFFSET FORMER_BP     ; Reservation of memory for lo_
                                                ; cal variables : 92 bytes. The
                                                ; stack can work normally below
                                                ; the area reserved for the local
                                                ; variables.

;......... Computing of BUFFER(i) = (XY)^i ;..................................

                FINIT                           ; Initialisation of the numeric
                                                ; data processor.
                XOR     DI,DI                   ; DI = 0
                MOV     CX,LENGTH BUFFER        ; CX = 10
                MOV     VL.X,TEST_X             ; [BP-OFFSET FORMER_BP].X =TEST_X
```

```
                                                ; (immediate value).  BP - OFFSET
                                                ; FORMER_BP,  working  implicitly
                                                ; with  SS, points the  beginning
                                                ; of the structured  area. Placed
                                                ; at the start, .X adds an offset
                                                ; equal to zero.
              FILD     VL.X                     ; ST(0)= integer. ST(0) is loaded
                                                ; from memory. The name of  field
                                                ; X gives the type WORD.
              FST      VL.X_REAL                ; ST(0) stored in X_REAL=Real short
              MOV      DX,TEST_Y
              FILD     VL.Y
              FST      VL.Y_REAL                ; ST(0) = Y_REAL         ST(1) = X

              FMULP    ST(1),ST(0)              ; ST(1) = ST(0)*ST(1) then ST(0)=
                                                ; ST(1).  At  the end ST(0) = X*Y
              FLD      ST(0)                    ; ST(0) = X*Y           ST(1) = X*Y
      NEXT:   FST      VL.BUFFER[DI]            ; BUFFER(i)  <=  (X*Y)ⁱ
              ADD      DI,TYPE BUFFER           ; DI = DI+8

              FMUL     ST,ST(1)                 ; ST(0) = (X*Y)ⁱ  ST(1) = X*Y
              LOOP     NEXT
;             .
;             .
              FINIT                             ; Flush the  stack of the  8087
              MOV      SP,BP                    ; The  area of local variables is
                                                ; returned to the stack.
              POP      BP                       ; Restoring of the former BP. The
                                                ; stack is in its initial state.
              RET
;
 PRODUCTS_XY  ENDP
;
  CODE_PREP  ENDS
;
              END
```

4.4.2 Parameter passing by the stack

In example 4.13, the values of X and Y are typically input values for the procedure.

Why not have them written on to the stack by the calling program? In this case only the structure is modified. This is the method that is almost exclusively used by the high level languages (see below).

Example 4.14

```
              NAME     MODULE_C
;             ===============

              EXTRN    PRODUCTS_XY:FAR

;
  CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; ===================================
          ASSUME  CS:CODE_PP

    START:
;             .
;             .                                 (Line C)
              PUSH     WORD PTR CS:VAL_X+2
              PUSH     WORD PTR CS:VAL_X
              PUSH     WORD PTR CS:VAL_Y+2
              PUSH     WORD PTR CS:VAL_Y        ; X and Y  are  "passed"  to  the
                                                ; procedure, in the form of short
                                                ; integer.
```

```
                CALL    PRODUCTS_XY             ; This called procedure is in
                                                ; charge of restoring the stack
                                                ; to the state existing in line C
                                                ; (before passing of parameters).
;               .
   VAL_X        DD      125325
   VAL_Y        DD      235000
;               .
;               .
  CODE_PP       ENDS
;
                END  START


                NAME    MODULE_A
;               ================

                PUBLIC  PRODUCTS_XY
;
   PARAM  STRUC
;  - - - - - - -
   BUFFER             DQ      10 DUP(?)         ; Local variables.
   FORMER_BP          DW      ?
   RETURN_ADDRESS     DD      ?                 ; FAR procedure, therefore CS and
                                                ; IP are saved : CS before IP.
   Y                  DD      ?                 ; Parameter X.
   X                  DD      ?                 ; Parameter Y.
   PARAM  ENDS
;
   P                  EQU     [BP-OFFSET FORMER_BP]
   AREA_PARAM         EQU     SIZE PARAM-OFFSET Y
;
  CODE_PREP   SEGMENT   BYTE   PUBLIC   'ROM'
; ====================================
       ASSUME  CS:CODE_PREP
;
  PRODUCTS_XY  PROC  FAR                        ; See example 4.12.
; =====================
                PUSH    BP
                MOV     BP,SP
                SUB     SP,OFFSET FORMER_BP
                FINIT
                XOR     DI,DI
                MOV     CX,LENGTH BUFFER

                FILD    P.X                     ; Loading of integer value X

                FILD    P.Y                     ; Loading of integer value Y
                                                ; ST(0) = Y     ST(1) = X
                FMULP   ST(1),ST(0)             ; ST(1)  =  ST(0) * ST(1)    then
                                                ; ST(1) stored in ST(0)
                FLD     ST(0)                   ; ST(0) = X*Y     ST(1) = X*Y

   NEXT:        FST     BUFFER[DI]
                ADD     DI,TYPE BUFFER
                FMUL    ST(0),ST(1)
                LOOP    NEXT
;               .
;               .
                FINIT
                MOV     SP,BP                   ; Restoring of  area  occuped  by
                                                ; BUFFER (i).
                POP     BP                      ; Restoring of the former BP.
                RET     AREA_PARAM              ; =RET 8 The return is done first
                                                ; by the return address, then the
                                                ; stack pointer is incremented by
                                                ; 8 bytes in order  to  free  the
                                                ; area occupied by the parameters:
```

```
                                        ; erasing the effect of the  PUSH
                                        ; instruction of the calling pro_
                                        ; gram.
;
 PRODUCTS_XY  ENDP
;
  CODE_PREP  ENDS
;
             END
```

4.4.3 Pascal-assembler link

Example 4.14 showed the transfer of two real simple precision values X and Y. In order to be able to transfer the value of other type variables, it is necessary to know the size, in bytes, occupied by each type.

For MS-Pascal (version 3.13), the sizes of the main types are as follows:

BOOLEAN	= 1 byte	(0 = false; 1 = true)
CHAR	= 1 byte	(ASCII code)
BYTE	= 1 byte	(unsigned binary)
WORD	= 2 bytes	(unsigned binary)
INTEGER	= 2 bytes	(signed binary)
INTEGER4	= 4 bytes	(signed binary)
REAL	= 4 bytes	(MS format)
REAL4	= 4 bytes	(standard IEEE format)
REAL8	= 8 bytes	(standard IEEE format)
AD OF...	= 2 bytes	(pointer containing the offset)
ADS OF...	= 4 bytes	(pointer containing the offset and the segment)
LSTRING(n)	= n bytes+1	(containing the length of the string)

There is also the facility for passing the address of a variable rather than its value. The advantage is clear for structured or long variables, so that their contents do not have to be copied on the stack.

In the case of procedures, this format is very important because it is the only one that makes it possible for the assembler to respond by writing the results to the addresses indicated by the calling program.

The transfer of this address can be done either in VAR, where only the offset is passed and the segment is implicitly the value contained in DS, or in VARS, where the offset and the segment are passed.

Examples 4.15, 4.15a, A.1 and A.1a illustrate this transfer of parameters by value and by address; A.1 and A.1a show how to return the address of a table defined in assembler.

Example 4.15

```
{$LINESIZE:131}
{$PAGESIZE:62}
!
!        This program calls for an assembler language procedure,  using  the  8087
!numeric data processor, in order to compute the square  root of the  sum of  the
!squares of an integer, of a real simple and a real double precision. The  result
!is returned by the assembler in double precision.
!
!        This program uses the  3.13 version of  MS_PASCAL which follows the  IEEE
!rule of numeric representation for the types REAL4 and REAL8. This rule is  that
!followed by the 8087. Previous versions only  have the type  REAL and  therefore
!cannot be used here.


                 PROGRAM   CALL_PROCEDURE (INPUT,OUTPUT);
!                ======================================

                 VAR     I:INTEGER;
                         R4:REAL4;
                         RESULT,R8:REAL8;

 PROCEDURE SQROOT(R8:REAL8;R4:REAL4;I:INTEGER;VARS RESULT:REAL8);EXTERN;

!
 BEGIN
!
         WRITE('I = ');READLN(I);
         WRITE('R4 = ');READLN(R4);
         WRITE('R8 = ');READLN(R8);
         SQROOT(R8,R4,I,RESULT);
         WRITELN('RESULT = ',RESULT);
!
 END.
!
```

Example 4.15a

```
PAGE 62,132
TITLE SQROOT
;
;        In addition to showing how the 8087 performs the required task, this sub_
;routine shows the automatic conversion of the different input or output  formats
;within the internal real temporary 80 bits format. The conversion depends on the
;type given,either explicitly or through the structure,  as well as  on the  ins_
;truction distinguishing integers from reals.


                NAME    SQROOT
;               ==============

                PUBLIC  SQROOT
;
    PARAM  STRUC
;   - - - - - - -
        DS_SYSTEM       DW      ?              ; Store for DS of calling program

        OLD_BP          DW      ?              ; BP points this level of the stack
        RETURN_ADD      DD      ?              ; Return address : IP then CS.
        RESULT_ADD      DD      ?              ; Address of RESULT
                                               ; offset then segment (VARS).
        VAL_I           DW      ?               ; Value of I (short integer).
        VAL_R4          DD      ?               ; Value of R4 (short real).
        VAL_R8          DQ      ?               ; Value of R8 (long real).

    PARAM  ENDS
;
        P               EQU     [BP-OFFSET OLD_BP]
        LENGTH_P        EQU     SIZE PARAM - (OFFSET RETURN_ADD+4)
                                                ; This last directive allows  the
                                                ; changing of name and the  orde_
                                                ; ring of the parameters  without
                                                ; changing the  instruction  RET.
                                                ; The same  method  is thus  used
                                                ; for  the  examples  4.15, 4.16,
                                                ; 4.17, 4.18.
   CODE_SQROOT   SEGMENT
;  ====================
          ASSUME  CS:CODE_SQROOT
;
  SQROOT  PROC  FAR
; ================

;.......Organisation of the stack and storage :...............................

                PUSH    BP                      ; Store for BP of calling program
                MOV     BP,SP                   ; Placing of the  stack  pointer.
                SUB     SP,OFFSET OLD_BP        ; Reservation  of  the  area  for
                                                ; local variables.
                MOV     P.DS_SYSTEM,DS          ; Storage of DS.

;.......Use of 8087. Automatic change of the formats ..........................

                FINIT                           ; Initialisation of  8087 ( stan_
                                                ; dard state).
                FILD    P.VAL_I                 ; ST(0) = value of I
                FMUL    ST,ST(0)                ; ST(0) = I²
                FLD     P.VAL_R4                ; ST(0) = value of R4, ST(1) = I²
                FMUL    ST,ST(0)                ; ST(0) = R4², ST(1) = I²
                FADDP   ST(1),ST                ; ST(1) =R4²+I² then ST(1) within
                                                ; ST(0).
                FLD     P.VAL_R8                ; Actual value of R8 within ST(0)
                                                ; ST(1) = R4² + I²
                FMUL    ST,ST(0)                ; ST(0) = R8², ST(1) = R4² + I²
                FADDP   ST(1),ST                ; ST(0) = R8² + R4² + I²
                FSQRT                           ; ST(0) = ( R8² + R4² + I² )0.5

;.......Writing of the result :...............................................

                LDS     BX,P.RESULT_ADD         ; DS:[BX] contains the address of
                                                ; RESULT.
                FSTP    QWORD PTR [BX]          ; DS:[BX] = ST(0)  in real  long.
                                                ; QWORD  type  must be  indicated
                                                ; since  it is an  anonymous  ad_
                                                ; dress. The 8087 stack is  empty
                                                ; after this  instruction.
;.......Restoring and return :................................................

                MOV     DS,P.DS_SYSTEM
                MOV     SP,BP                   ; Freeing  of the  area of  local
                                                ; variables.
                POP     BP
                RET     LENGTH_P                ; Return then freeing of the area
                                                ; of parameters.
  SQROOT  ENDP
;
  CODE_SQROOT   ENDS
;
                END
```

On the other hand, in the case of functions, return is always carried out by value.

In the case of integers, boolean or real numbers (MS-Pascal), the transfer is simply made by the CPU registers.

* 1 byte: the result is written in AL.
  AH is set to 0.

* 2 bytes: the result is written in AX.
  AH contains the high order bit and AL the low order bit.

* 4 bytes: (REAL, pointer) the result is written in ES:[BX]. ES contains the high order word and BX the low order word.

* 4 bytes and more: (REAL4, REAL8, structured variables) the result is written in a temporary location in the stack. This location is set by the Pascal program and passed to the assembler in the form of an offset written after the parameters and before the new branch address. On the return, the assembler must recopy this offset in the accumulator. The segment of this temporary location is always held in SS since the stack is involved.

Examples 4.16 and 4.16a use these formats for the calling of a REAL8 type function, while A.1 and A.1a show the return of a boolean number in a function of this type, placed directly in an IF instruction.

Example 4.16

```
{$LINESIZE:131}
{$PAGESIZE:62}
!
!       This program executes the same task as the one defined in  example  4.15,
!but the assembler  procedure is  here a  function of the  PASCAL  program  (3.13
!version of MS-PASCAL).


                PROGRAM   CALL_FUNCTION(INPUT,OUTPUT);
!               ======================================

                 VAR     I:INTEGER;
                         R4:REAL4;
                         RESULT,R8:REAL8;

                 FUNCTION SQROOT(R8:REAL8;R4:REAL4;I:INTEGER):REAL8;EXTERN;


!
```

```
              BEGIN
!
              WRITE('I = ');READLN(I);
              WRITE('R4 = ');READLN(R4);
              WRITE('R8 = ');READLN(R8);
              WRITELN('RESULT = ',SQROOT(R8,R4,I));
!
              END.
!
```

Example 4.16a

```
PAGE 62,132
TITLE SQFNCP
;
;       This program  executes the same  task as the one in example  4.15, but it
;answers as a function by  writing the  result in a  temporary  variable  created
;within the stack by the calling program.
;       See also commentaries on the 4.15A example.


              NAME    SQFNCP
;             ==============

              PUBLIC  SQROOT
;
  PARAM  STRUC
; - - - - - - - -
        DS_SYSTEM       DW      ?
        FORMER_BP       DW      ?
        RETURN_ADD      DD      ?
        RESULT_OFF      DW      ?               ; Offset  of  temporary  variable
                                                ; stored in the stack.
        VAL_I           DW      ?
        VAL_R4          DD      ?
        VAL_R8          DQ      ?
;
  PARAM  ENDS
;
        P               EQU     [BP-OFFSET FORMER_BP]
        LENGTH_P        EQU     SIZE PARAM - (OFFSET RETURN_ADD+4)
;
   CODE_SQROOT   SEGMENT
;  ===================
         ASSUME  CS:CODE_SQROOT
;
  SQROOT  PROC  FAR
; ----------------

;.......Organisation of the stack and storage :................................

              PUSH    BP
              MOV     BP,SP
              SUB     SP,OFFSET FORMER_BP

              MOV     P.DS_SYSTEM,DS
```

```
;.......Use of 8087 :.........................................................

             FINIT

             FILD    P.VAL_I
             FMUL    ST,ST(0)
             FLD     P.VAL_R4
             FMUL    ST,ST(0)
             FADDP   ST(1),ST
             FLD     P.VAL_R8
             FMUL    ST,ST(0)
             FADDP   ST(1),ST
             FSQRT
;

;.......Writing of result :...................................................

             MOV     BX,P.RESULT_OFF          ; BX = offset within the stack of
                                              ; the temporary variable.
             FSTP    QWORD PTR SS:[BX]        ; Writing in SS:[BX], with  indi_
                                              ; cation of the type.
             MOV     AX,BX                    ; Copy  in  AX of the  offset  of
                                              ; temporary variable.

;.......Restoring and return :.................................................

             MOV     DS,P.DS_SYSTEM
             MOV     SP,BP
             POP     BP
             RET     LENGTH_P
;
  SQROOT  ENDP
;
   CODE_SQROOT  ENDS
;
             END
```

In all cases of procedures or functions, the branching is of type FAR and the called program must preserve the value of DS and restore it intact on return.

It is equally more prudent to do the same for BP.

The RET instruction must ensure the clearing of locations occupied by the passing of parameters.

4.4.4 FORTRAN-assembler link

For MS-FORTRAN the size of possible values is:

Integer	:2 bytes	(INTEGER *2)
Integer long	:4 bytes	(INTEGER *4)
Real short	:4 bytes	(REAL * 4)
Real double precision	:8 bytes to IEEE standard	(REAL * 8)
Boolean	:2 bytes	(LOGICAL * 2)
Boolean long	:4 bytes	(LOGICAL * 4)

The assembler subroutine is called by MS-FORTRAN with the aid of a FAR type branch.

The subroutine is charged with the task of conserving the values of the BP, DS and SS registers.

These values must be restored on return.

The RET instruction must also ensure the clearing of the location occupied by the passing of parameters.

The FORTRAN Program passes the parameter addresses, with the aid of two-word pointers, and not their values. (Equivalent to the VARS of MS-Pascal.)

In procedures, results are written by the assembler subroutine to the addresses indicated.

On the other hand, in the case of functions the return is always carried out by value.

In the case of integers and booleans, the transfer is done simply by the CPU registers:

2 bytes (integer, boolean)
the result is written in AX by the assembler;
AH contains the high order word and AX the low order word.

4 bytes (long integer, long boolean)
the result is written in DX : AX;
DX contains the high order word and AX the low order word.

In the case of real numbers (short in 4 bytes or long in 8 bytes), a location is reserved in the stack, by the FORTRAN program, for receiving the value of the result. It is this location address that is passed to the assembler. Only the offset of this address is necessary in the stack, since its segment value is contained implicitly in SS.

This offset is written to the stack after the parameter pointers and just before the return address pointer.

Examples 4.17 and 4.17a show the calling of a procedure, while examples 4.18 and 4.18a show the use of a function.

Example 4.17

```
C
C
C               This program calls an assembler procedure using the 8087
C       numeric data processor to compute the square root of the sum  of
C       the squares of an integer, of a real  simple and  a real  double
C       precision. The result is returned  by the  assembler  in  double
C       precision.
C
C               This program uses the 3.13 version of  MS-FORTRAN  which
C       follows the IEEE  rule of numeric  representation for the  types
C       REAL*4 and REAL*8. The 8087 follows this rule.
C
C
C
```

```
        REAL*8 RESULT,R8
        WRITE(*,500)
        READ(*,100)I
        WRITE(*,600)
        READ(*,200)R4
        WRITE(*,700)
        READ(*,300)R8
        CALL SQFOR(R8,R4,I,RESULT)
        WRITE(*,400)RESULT
100     FORMAT(I8)
200     FORMAT(E5.2)
300     FORMAT(D5.2)
400     FORMAT(1X,'RESULT = ',D10.4)
500     FORMAT(1X,'I = ')
600     FORMAT(1X,'R4 = ')
700     FORMAT(1X,'R8 = ')
        STOP
        END
```

Example 4.17a

```
PAGE  62,132
TITLE  SQFOR
;
;       This subroutine is called by the FORTRAN program to perform the same task
;as that of example 4.15A.
;Note that as Fortran only passes the addresses of variables as parameters (equi_
;valent to VARS of MS_PASCAL), reading and writing of parameters is done indirec_
;tly, thus needing the explicit indication of the type contained in the  transfer
;instruction.
;       See also commentaries on the 4.15A example.


                NAME    SQFOR
;               ============

                PUBLIC  SQFOR
;
    PARAM  STRUC
;   - - - - - - -
        DS_SYSTEM       DW      ?
        FORMER_BP       DW      ?
        RETURN_ADD      DD      ?
        RESULT_ADD      DD      ?              ; Segment and offset of variables:
        ADD_I           DD      ?              ; R8, R4, I and RESULT in the wri_
        ADD_R4          DD      ?              ; ting order of the procedure call
        ADD_R8          DD      ?
;
    PARAM  ENDS
;
        P               EQU     [BP-OFFSET FORMER_BP]
        LENGTH_P        EQU     SIZE PARAM - (OFFSET RETURN_ADD+4)
;
   CODE_SQFOR   SEGMENT
;  =================
         ASSUME  CS:CODE_SQFOR
;
  SQFOR  PROC  FAR
; ---------------

;.......Organisation of the stack and storage :................................

                 PUSH    BP
                 MOV     BP,SP
                 SUB     SP,OFFSET FORMER_BP

                 MOV     P.DS_SYSTEM,DS
;
```

```
;.......Use of 8087. Automatic change of the formats :.........................

                FINIT

                LDS     BX,P.ADD_I              ; DS:[BX] contains the address
                                                ; of I.
                FILD    WORD PTR [BX]           ; ST(0) = value of I.
                FMUL    ST,ST(0)
                LDS     BX,P.ADD_R4             ; DS:[BX] = address of R4.
                FLD     DWORD PTR [BX]          ; ST(0) = value of R4.
                FMUL    ST,ST(0)
                FADDP   ST(1),ST
                LDS     BX,P.ADD_R8             ; DS:[BX] = address of R8.
                FLD     QWORD PTR [BX]          ; ST(0) = value of R8.
                FMUL    ST,ST(0)
                FADDP   ST(1),ST
                FSQRT

;.......Writing of result :....................................................

                LDS     BX,P.RESULT_ADD         ; DS:[BX] = address of RESULT.
                FSTP    QWORD PTR [BX]          ; Value of RESULT = ST(0). The
                                                ; stack is empty after this ins_
                                                ; truction.

;.......Restoring and return :.................................................


                MOV     DS,P.DS_SYSTEM
                MOV     SP,BP
                POP     BP
                RET     LENGTH_P
;
 SQFOR  ENDP
;
  CODE_SQFOR   ENDS
;
               END
```

Example 4.18

```
C
C
C       CALLING AN ASSEMBLER FUNCTION
C       Same tasks as in example 4.17.
C       SQROOT is declared in double precision.
C
C
        REAL*8 R8,SQROOT
        WRITE(*,500)
        READ(*,100)I
        WRITE(*,600)
        READ(*,200)R4
        WRITE(*,700)
        READ(*,300)R8
        WRITE(*,400)SQROOT(R8,R4,I)
100     FORMAT(I8)
200     FORMAT(E5.2)
300     FORMAT(D5.2)
400     FORMAT(1X,'RESULT = ',D10.4)
500     FORMAT(1X,'I = ')
600     FORMAT(1X,'R4 = ')
700     FORMAT(1X,'R8 = ')
        STOP
        END
```

Example 4.18a

```
PAGE 62,132
TITLE SQROOT
;
;       This procedure returns, to the calling program, a double  precision  type
;value.
;
;       A temporary variable is created within the stack and its offset is  writ_
;ten as a last parameter at the moment the function is called  ( identical  func_
;tionning for FORTRAN and PASCAL).
;
;       See also the commentaries on examples 4_15A and 4_17A.


                NAME    SQROOT
;               ==============

                PUBLIC  SQROOT
;
  PARAM  STRUC
; - - - - - - - -
         DS_SYSTEM       DW      ?
         FORMER_BP       DW      ?
         RETURN_ADD      DD      ?
         RESULT_OFF      DW      ?               ; Offset of the temporary  varia_
                                                 ; ble within the stack.
         ADD_I           DD      ?
         ADD_R4          DD      ?
         ADD_R8          DD      ?
;
  PARAM  ENDS
;
         P               EQU     [BP-OFFSET FORMER_BP]
         LENGTH_P        EQU     SIZE PARAM - (OFFSET RETURN_ADD+4)
;
  CODE_SQROOT  SEGMENT
; ===================
         ASSUME  CS:CODE_SQROOT
;
  SQROOT  PROC  FAR
; ----------------

;.......Organisation of the stack :...........................................

                PUSH    BP
                MOV     BP,SP
                SUB     SP,OFFSET FORMER_BP

                MOV     P.DS_SYSTEM,DS
;

;.......Use of 8087 :.........................................................

                FINIT

                LDS     BX,P.ADD_I
                FILD    WORD PTR [BX]
                FMUL    ST,ST(0)
                LDS     BX,P.ADD_R4
                FLD     DWORD PTR [BX]
                FMUL    ST,ST(0)
                FADDP   ST(1),ST
                LDS     BX,P.ADD_R8
                FLD     QWORD PTR [BX]
                FMUL    ST,ST(0)
                FADDP   ST(1),ST
                FSQRT
```

```
;.......Result issue :....................................................

                MOV     BX,P.RESULT_OFF         ; BX = offset  of  the  temporary
                                                ; variable.
                FSTP    QWORD PTR SS:[BX]       ; Writing in SS:[BX],  with  type
                                                ; indication.

;.......Restoring and return :............................................

                MOV     DS,P.DS_SYSTEM
                MOV     SP,BP
                POP     BP
                RET     LENGTH_P
;
  SQROOT  ENDP
;
   CODE_SQROOT   ENDS
;
               END
```

4.4.5 BASIC 86-assembler link

For BASIC, assembler language procedures are called by means of a CALL (formerly USR) statement. This branching is always of a far type (FAR).

Two methods of operation must be distinguished, as follows.

Compiled mode

This method of operation is very similar to that of Pascal or FORTRAN. The object modules coming from the BASIC or from the assembler are joined by the LINK utility and the program *.EXE is loaded by the system for execution.

In the BASIC program the CALL order simply refers to the name of the assembler procedure.

This mode has been used in examples 4.19 and 4.19a.

Interpreted mode

Here the method of operation is less convenient because the assembler module must be loaded into memory at the same time as the BASIC program and its interpreter. In addition, the procedure address must be known to the programmer.

Different solutions, often scarcely practical, are offered depending on the system environment. The user needs to refer to his own particular BASIC User´s Guide.

Apart from this loading problem, there are no differences between the two modes.

The size of the variables is as follows:

Integer	2 bytes
Real, simple precision	4 bytes (Microsoft, non-IEE)
Real, double precision	8 bytes (Microsoft)

Examples 4.21 and 4.21a use two short procedures to transform the MS-DOS representation of a real into the IEEE form, and vice versa.

For character strings, the MS-BASIC compiler uses a string descriptor consisting of 4 bytes. The first two bytes give the string length in 16 bits, restricted to 32767. The second two bytes give the offset of the beginning of the string. When a string of characters is used as a parameter, the offset of this descriptor is indicated to the subroutine.

Examples 4.20 and 4.20a illustrate the use of this descriptor in order to effect the exchange of a string of characters between BASIC and assembler, and vice versa.

The BASIC program passes the parameters by means of their single offset address. The segment used is the one held in DS (equivalent to VAR in MS-Pascal).

These parameter offsets are pushed on to the stack in the order in which they are written in the call.

If a particular stack is used by the assembler, the source stack must of course be restored before the return, and the RET instruction must ensure the clearing of the locations occupied during the parameter transfers.

Example 4.19

```
50  '
60  '            This program performs a byte by byte reception on the port AUX, by  means
70  'of an assembler call, and puts the bytes read into file.
80  '
90  '            After reception, the same file may also be transmitted  through the  port
100 'AUX.
110 '
120 '
130 '......Reception :.........................................................
140 '
150 OPEN "O",1,"FILE.LOW"
160 PRINT "RECEPTION ENDS WITH alt Z"
170 PRINT "FOR TEXT THE END OF LINE IS : alt M,alt J"
180 '
190 CALL RECEPTION(ENTRY%,TERM%)
200 IF TERM%=1 THEN 190
210 WRITE   ,ENTRY%
220 GOTO 140
230 '
240 WRITE   ,ENTRY%
250 CLOSE 1
260 PRINT
270 INPUT "RECEPTION ENDED, TRANSMIT ?<Y/N> ";ANSWER$
280 IF ANSWER$<>"Y" THEN 340
290 '
300 '.....Transmission :.......................................................
310 '
320 OPEN "I",1,"FILE.LOW"
330 '
340 IF EOF(1) THEN 340
350 INPUT   ,EXIT%
```

```
360 CALL TRANSMIT(EXIT%)
370 GOTO 290
380 '
390 PRINT
400 PRINT "TRANSMISSION ENDED"
410 END
```

Example 4.19a

```
PAGE 62,132
TITLE LINK
;
;       This subroutine uses the AUX input and output  system  calls, a  standard
;usually associated with RS 232 serial port. Through this port it is possible  to
;establish a connection between two systems in  order to  exchange  textual  data
;(e.g. language source).
;
;       This program reads and writes the bytes contained within the  lower  half
;of a BASIC integer.
;
;       In  order to  trace the  transmissions, these  bytes are  written on  the
;screen in hexadecimal form.This translation is  performed with  the aid  of  the
;instruction XLAT.
;
;       The end of file symbol here is ALT-Z (26 in ASCII) which is most frequen_
;tly used, but it can easily be modified if required.
;
;       The function of the  procedure  is  simulated  by  reading  the  keyboard
;(without echo) and by screen writing. In order to work  with the  AUX  port, the
;directives EQU for the AUX_ENTRY and AUX_EXIT must be modified.
;
;       The procedure assumes either version 1.25a or 2.11 of MS_DOS.


                NAME    LINK
;               ============

                PUBLIC  TRANSMISSION,RECEPTION

;
    RECEP  STRUC
;   - - - - - - -
    FORMER_BP           DW      ?
    RETURN_ADD          DD      ?
    END_OFF             DW      ?
    ENTRY_OFF           DW      ?
;
    RECEP  ENDS
;
    R                   EQU     [BP - OFFSET FORMER_BP]
    LENGTH_R            EQU     SIZE RECEP - (OFFSET RETURN_ADD + 4)
                                        ; Writing  allowing  changes   of
                                        ; parameters, as desired.
    EMI  STRUC
;   - - - - - -
    OLD_BP              DW      ?
    RETURN              DD      ?
    EXIT_OFF            DW      ?
;
    EMI  ENDS
;

    E                   EQU     [BP - OFFSET OLD_BP]
    LENGTH_E            EQU     SIZE EMI - (OFFSET RETURN + 4)
                                        ; Writing  allowing  changes   of
                                        ; parameters, as desired.
```

```
  RECEPTION_END        EQU     26
  MS_DOS_CALL          EQU     21H
  AUX_ENTRY            EQU     08H              ; Simulated by  keyboard  call
                                                ; without echo. Must be  replaced
                                                ; by 03 for AUX.
  AUX_EXIT             EQU     02H              ; Simulated  by  screen  display.
                                                ; Must be replaced by 04 for AUX.
  SCREEN_VIEW          EQU     02H
;
  CODE_LINK   SEGMENT
; ==================
       ASSUME  CS:CODE_LINK

  TRANSLATE_TABLE      LABEL   BYTE
                       DB      '0123456789ABCDEF' ; List of ASCII codes of hexa_
                                                ; decimal numbers from 0 to F,or_
                                                ; dered according to their values.
 DISPLAY     PROC  NEAR                         ; Screen  display of the  content
; --------------------                          ; of AL in hexadecimal form.

;.......Storage of registers used :...........................................

               PUSH    DX
               PUSH    CX
               PUSH    BX
               PUSH    AX
               MOV     DH,AL                    ; Storage of AL in AH.

;.......Loading of the table offset of translation in BX :.....................

               LEA     BX,TRANSLATE_TABLE

;.......Right shift in order to isolate upper part number :....................

               MOV     CL,4                     ; 4 right  shifts, 0000 on  upper
               SHR     AL,CL                    ; part.

;.......Translation of the upper part in ASCII code :..........................

                XLAT    TRANSLATE_TABLE         ; The value of the table  addres_
                                                ; sed by BX and indexed by AL  is
                                                ; transferred to  AL.  TRANSLATE_
                                                ; TABLE is used only  for  giving
                                                ; the type (byte) and the segment
                                                ; (CS).
;.......Display of the upper part number :.....................................

               MOV     DL,AL
               MOV     AH,SCREEN_VIEW
               INT     MS_DOS_CALL
;

;.......Display of the lower part number :.....................................

               MOV     AL,DH                    ; Restoring of AL.
               AND     AL,0FH                   ; The lower  number is  isolated.

               XLAT    TRANSLATE_TABLE          ; Translation in ASCII code.

               MOV     DL,AL
               MOV     AH,SCREEN_VIEW
               INT     MS_DOS_CALL
```

```
;.......Restoring of registers used and return :..............................

                PUSH    AX
```

```
;.......Restoring of registers used and return :..............................

                POP     AX
                POP     BX
                POP     CX
                POP     DX

                RET
;
  DISPLAY    ENDP

  RECEPTION  PROC  FAR
; --------------------

;.......Ordering of the stack (without local variables) :......................

                PUSH    BP
                MOV     BP,SP

;.......Reading of offsets of BASIC variables :................................

                 MOV     BX,R.ENTRY_OFF
                 MOV     DI,R.END_OFF

;.......Reception of a byte :..................................................

                MOV     AH,AUX_ENTRY
                INT     MS_DOS_CALL
                PUSH    AX

;.......Display of received byte, followed by two blanks :.....................

                CALL    DISPLAY
                MOV     DL,' '
                MOV     AH,SCREEN_VIEW
                INT     MS_DOS_CALL
                MOV     DL,' '
                MOV     AH,SCREEN_VIEW
                INT     MS_DOS_CALL

                POP     AX

;.......Eventual detection of end of transmission (alt_Z) :....................

                CMP     AL,RECEPTION_END
                MOV     WORD PTR [DI],0         ; END = 0 a priori.
                JNE     NOT_END
                MOV     WORD PTR [DI],1         ; END = 1  if   RECEPTION_END  is
                                                ; found.

;.......Sending bytes to BASIC :...............................................

   NOT_END:     XOR     AH,AH
                MOV     [BX],AX                 ; The end of  reception  byte  is
                                                ; also sent.

;.......Restoring of the stack and return :....................................

                MOV     SP,BP
                POP     BP
                RET     LENGTH_R                ; "clearing " of parameters.

  RECEPTION  ENDP
  TRANSMISSION  PROC  FAR
; ----------------------

;.......Ordering of stack (without local variables) :...........................

                PUSH    BP
                MOV     BP,SP

;.......Reading of the offset then of the contents of the BASIC variable :........

                MOV     BX,E.EXIT_OFF

                MOV     AL,[BX]
                PUSH    AX

;.......Screen display of the contents in hexadecimal followed by ">" :..........

                CALL    DISPLAY

                MOV     DL,'>'
                MOV     AH,SCREEN_VIEW
                INT     MS_DOS_CALL

;.......Sending of the byte to AUX :..........................................

                POP     DX
                MOV     AH,AUX_EXIT
                INT     MS_DOS_CALL

;.......Sending of the separating blank :......................................

                MOV     DL,' '
                MOV     AH,SCREEN_VIEW
                INT     MS_DOS_CALL

;.......Restoring of the stack and return :....................................

                MOV     SP,BP
                POP     BP
                RET     LENGTH_E                ; "clearing" of parameters.

  TRANSMISSION  ENDP
;
   CODE_LINK  ENDS
;
                END
```

Example 4.20

```
80 '            This example  shows the  exchange of a character string  between a  BASIC
90 'program and an assembler subroutine.
100'
110'            The calling program gives the offset of the address of a describer  which
120'contains the length and the offset of location of the string.
130'
140'            The subroutine displays this  character string on the screen, in  between
150'angle brackets. Then, it answers by overwriting the  memory  locations  used  by  the
160'BASIC string with an  answer. In  this  case,  the  procedure  must  not  write  more
170'characters back than were initially contained in the BASIC string  in order to  avoid
180'erasing part of the calling programs instructions.
190'
200'
210 INPUT "STRING TO SEND";CHAIN$
220 CALL DISPLAY(CHAIN$)
230 PRINT CHAIN$
240 INPUT "NEW TRIAL ? <Y/N> ";ANSWER$
250 IF ANSWER$="Y" THEN GOTO 200
260 END
```

Example 4.20a

```
        PAGE     62,132
        TITLE    DISPLAY_STRING
;
                        NAME    DISPLAY_STRING
;                       ======================

                         PUBLIC  DISPLAY

                SCREEN_DISPLAY  EQU     2
                MS_DOS_CALL     EQU     21H
;
         PARAM  STRUC
;        ============

         FORMER_BP       DW      ?
         ADD_RETURN      DD      ?
         OFF_DESCRIBER   DW      ?
;
         PARAM  ENDS
;
         P               EQU     [BP - OFFSET FORMER_BP]
         DEL_PARAM       EQU     SIZE PARAM - (OFFSET ADD_RETURN + 4)
;
         DESCRIBER  STRUC
;        ================

         LONG_STRING     DW      ?
         OFF_STRING      DW      ?
;
         DESCRIBER  ENDS
;
 CODE_DISPLAY  SEGMENT  PUBLIC
;*****************************
               ASSUME          CS:CODE_DISPLAY

         ANSWER          DB      ODH,OAH,'TRANSMISSION IS OK'    ; ODH and OAH are counted
         END_ANSWER      LABEL   BYTE                            ; in the string length.
;
 DISPLAY  PROC  FAR
;=================

;..............Ordering of the stack (without local variables) :......................

                         PUSH    BP
                         MOV     BP,SP

;..............Reading of the BASIC character string describer :......................

                         MOV     BX,P.OFF_DESCRIBER      ; BX contains offset of the BASIC
                                                         ; character string describer.
                         MOV     CX,[BX].LONG_STRING     ; CX = length of string.
                         JCXZ    END_WORK                ; Abort if the string is empty.
                         MOV     DI,CX                   ; Temporary storage of CX.
                         MOV     BX,[BX].OFF_STRING      ; BX = offset of the string.

;..............Display of the string of characters between <> :......................

                         MOV     AH,SCREEN_DISPLAY
                         MOV     DL,'<'
                         INT     MS_DOS_CALL

                         XOR     SI,SI

         DISP_LOOP:      MOV     DL,[BX+SI]
                         INT     MS_DOS_CALL
                         INC     SI
                         LOOP    DISP_LOOP
```

```
                          MOV     DL,'>'
                          INT     MS_DOS_CALL

;..............Overwriting of the answer on the BASIC character string ;................

                          MOV     CX,DI                   ; Restoring: CX= length of string
                          XOR     SI,SI

          ANSWER_LOOP:    MOV     AL,ANSWER[SI]
                          MOV     [BX+SI],AL
                          INC     SI
                          CMP     SI,END_ANSWER - ANSWER
                          LOOPNE  ANSWER_LOOP             ; Looping until the end of answer
                                                          ; IF THE BASIC STRING IS LONG
                                                          ; ENOUGH.

                          JCXZ    END_WORK
          BLANKING:       MOV     BYTE PTR [BX+SI],' '    ; Clearing of the rest of the
                                                          ; BASIC character string.
                          INC     SI
                          LOOP    BLANKING
          END_WORK:

;..............Restoring of the stack and return ;...................................

                          MOV     SP,BP
                          POP     BP
                          RET     DEL_PARAM
;
 DISPLAY  ENDP
;
 CODE_DISPLAY   ENDS
;
                          END
```

Example 4.21

```
80  '          This example shows the exchange of simple precision real value
90  'between a BASIC program and an assembler subroutine,which is named "ROOT".
100 '
110 '          The calling program gives the offset of the address of two real
120 'numbers. The second parameter "OUTREAL" is written, by the "ROOT" assembler
130 'subroutine, with the square root of the first parameter "INREAL".
130 '
140 '          This result is then displayed on the screen by a PRINT instruction.
150 '
160 INPUT "NUMBER TO COMPUTE";INREAL
170 PRINT " THE SQUARE ROOT OF ";INREAL;" IS ";
180 CALL ROOT(INREAL,OUTREAL)
190 PRINT OUTREAL
200 INPUT "NEW TRIAL ? <Y/N> ";ANSWER$
210 IF ANSWER$="Y" GOTO 160
220 END
```

Example 4.21a

```
        PAGE 62,132
        TITLE ROOT_WITH_BASIC
;
;               The BASIC calling  program gives the  offset of the  address of two  real
;numbers. The first one is converted to IEEE numeric notation, then it is loaded into the
;8087 numeric data processor. The result of square root, which is computed  by the  8087,
;is then written into the second BASIC real number.
;               Finally, this number is  converted to Microsoft  numeric notation  before
;display on the screen by a PRINT instruction.
;
;                      IEEE data numeric representation:
;                      =================================
;
;    <1><------- 8 -----> <---------------------- 23 ----------------------->
;      S E E E E E E E   E M M M M M M M   M M M M M M M M   M M M M M M M M
;          byte  3           byte  2           byte  1           byte  0
;
;                      with     S = sign of number
;                               E = exponent with an offset of 127
;                               M = mantissa
;
;
;                   MICROSOFT data numeric representation:
;                   ======================================
;
;       <----- 8 ----->  <1><------------------ 23 ------------------------->
;       E E E E E E E E   S M M M M M M M  M M M M M M M M   M M M M M M M M
;           byte  3           byte  2          byte  1           byte  0
;
;                      with     E = exponent with an offset of 129
;                               S = sign of number
;                               M = mantissa
;
                        NAME    ROOT_WITH_BASIC
;                       =======================

                        PUBLIC  ROOT
;
        PARAM  STRUC
;       ============

        FORMER_BP       DW      ?
        RETURN_ADD      DD      ?
        OUTPUT_REAL_OFF DW      ?
        INPUT_REAL_OFF  DW      ?
;
        PARAM  ENDS
;
        P               EQU     [BP - OFFSET FORMER_BP]
        DELETE_PARAM    EQU     SIZE PARAM - (OFFSET RETURN_ADD + 4)
;

 CODE  SEGMENT  PUBLIC
;**********************
               ASSUME  CS:CODE
;
        NORM  PROC
;       ==========
;..........Conversion: MICROSOFT representation to IEEE representation :...............

                        CMP     BYTE PTR [BX+3],0       ; No  modification if the  number
                        JE      ZERO_ABORT              ; is equal to 0 (exponent = 0).

                        SUB     BYTE PTR [BX+3],2
                        RCL     BYTE PTR [BX+2],1
                        RCR     BYTE PTR [BX+3],1
                        RCR     BYTE PTR [BX+2],1
            ZERO_ABORT:
                        RET
```

```
;
        NORM  ENDP

        UNNORM  PROC
;       ===========
;..........Conversion: IEEE representation to MICROSOFT representation :..............

                        PUSH    AX
                        MOV     AX,[BX]+2               ; No  modification if the  result
                        SHL     AX,1                    ; is equal to 0 (exponent = 0).
                        CMP     AH,0
                        POP     AX
                        JE      ZERO

                        RCL     BYTE PTR [BX+2],1
                        RCL     BYTE PTR [BX+3],1
                        RCR     BYTE PTR [BX+2],1
                        ADD     BYTE PTR [BX+3],2
                        RET
           ZERO:
                        MOV     BYTE PTR [BX+3],0       ; If the  number is equal to  -0,
                                                        ; the sign is reset to 0 (+).
                        RET
;
        UNNORM  ENDP
;
        ROOT  PROC  FAR
;       ===============
;..............Ordering of the stack (without local variables) :.......................

                        PUSH    BP
                        MOV     BP,SP

                        FINIT                           ; Initialisation of the 8087.

;..............Conversion and loading of the input number :...........................

                        MOV     BX,P.INPUT_REAL_OFF
                        CMP     BYTE PTR [BX]+3,0       ; Is the number equal to 0 ?
                        JE      NUL

                        CALL    NORM
                        FLD     DWORD PTR [BX]

;..............Computation of the square root :.......................................

                        FSQRT
                        JMP     ISSUE

           NUL:         FLDZ                            ; 0 is loaded in ST(0).

;..............Issuing and inverse conversion of the result :.........................

           ISSUE:       MOV     BX,P.OUTPUT_REAL_OFF
                        FSTP    DWORD PTR [BX]
                        WAIT                            ; To be  sure that the  number is
                                                        ; completely written.
                        CALL    UNNORM

;..............Restoring of the stack and return :....................................

                        MOV     SP,BP
                        POP     BP

                        RET     DELETE_PARAM
;
        ROOT  ENDP
;
 CODE   ENDS
;
                        END
```

5 Circuit Description

5.1 Description of CPU

The 8086 and the 8088 are packaged as 40-pin devices, requiring a single 5 V power supply.

Both microprocessors are entirely compatible from the software point of view: a program executed by one can be executed by the other. The essential difference lies in the fact that the 8088 is only able to make 8-bit accesses to memory and peripherals instead of the 16-bit accesses of the 8086. As a result, programs take longer to execute on the 8088.

Each has two possible modes, selected by the logical state of pin 33. At 0, the microprocessor is in maximum mode, and at 1 it is in minimum mode.

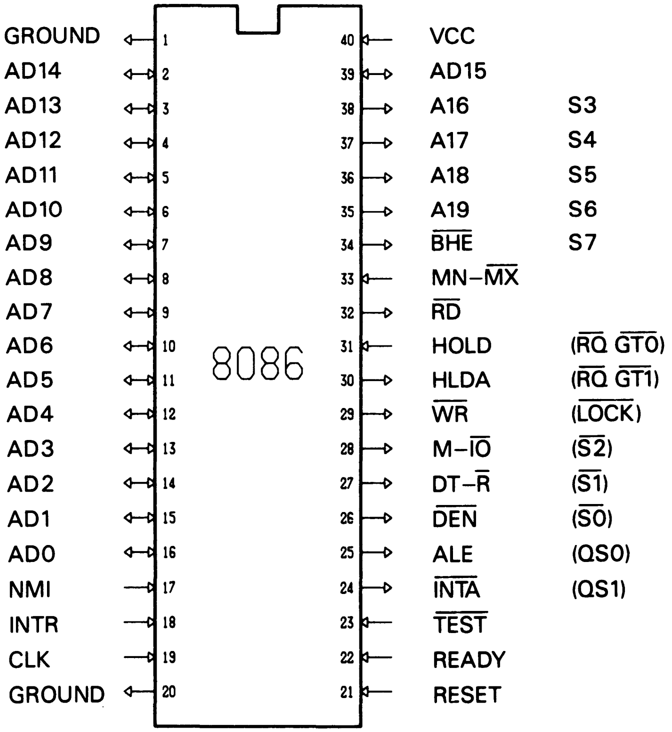

8086
GROUND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GROUND 20
40 VCC
39 AD15
38 A16 S3
37 A17 S4
36 A18 S5
35 A19 S6
34 $\overline{BHE}$ S7
33 MN-$\overline{MX}$
32 $\overline{RD}$
31 HOLD ($\overline{RQ}$ $\overline{GT0}$)
30 HLDA ($\overline{RQ}$ $\overline{GT1}$)
29 $\overline{WR}$ ($\overline{LOCK}$)
28 M-$\overline{IO}$ ($\overline{S2}$)
27 DT-$\overline{R}$ ($\overline{S1}$)
26 $\overline{DEN}$ ($\overline{S0}$)
25 ALE (QS0)
24 $\overline{INTA}$ (QS1)
23 $\overline{TEST}$
22 READY
21 RESET

Figure 5.1 8086

J.-M. Trio, *8086–8088 Architecture and Programming*
© English Language edition, Macmillan Publishers Ltd 1985

Minimum mode

This mode allows the number of external circuits required to be reduced to a minimum for small, single processor systems.

In this mode, the CPU retains its full addressing capacity and directly drives the control bus.

Maximum mode

This mode extends the architecture of the system to permit multiprocessor configurations and thus enables the CPU instruction set to be extended to include the numeric processor instructions.

The provision of two levels of priority for bus requests allow several processors to reside on the local bus of the 8086 and share its interfacing with the system bus.

5.1.1 Pin assignment

A brief description of the pin assignment follows below. The functions that correspond to the various pins will be described in more detail throughout this chapter. A bar placed over the pin name indicates that

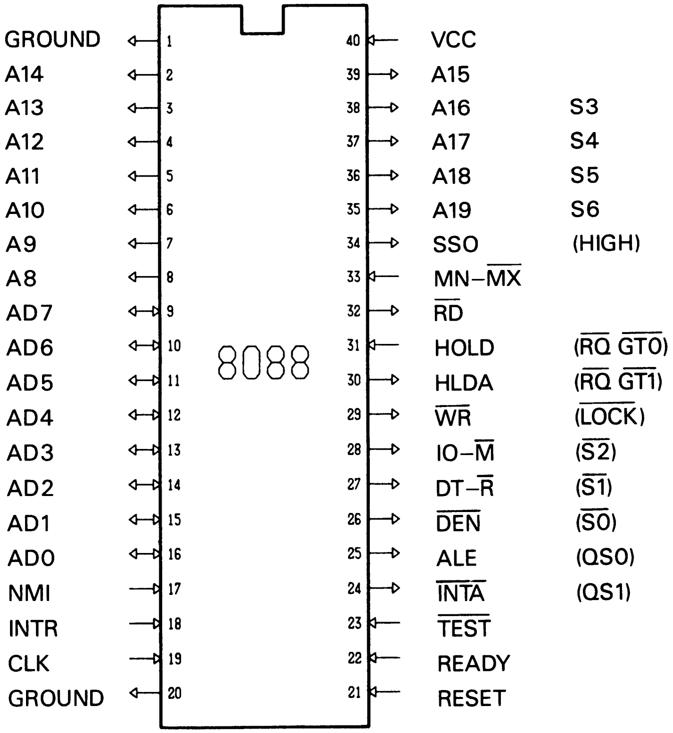

8088
GROUND 1
A14 2
A13 3
A12 4
A11 5
A10 6
A9 7
A8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GROUND 20
40 VCC
39 A15
38 A16 S3
37 A17 S4
36 A18 S5
35 A19 S6
34 SSO (HIGH)
33 MN-$\overline{\text{MX}}$
32 $\overline{\text{RD}}$
31 HOLD ($\overline{\text{RQ}}$ $\overline{\text{GT0}}$)
30 HLDA ($\overline{\text{RQ}}$ $\overline{\text{GT1}}$)
29 $\overline{\text{WR}}$ ($\overline{\text{LOCK}}$)
28 IO-$\overline{\text{M}}$ ($\overline{\text{S2}}$)
27 DT-$\overline{\text{R}}$ ($\overline{\text{S1}}$)
26 $\overline{\text{DEN}}$ ($\overline{\text{S0}}$)
25 ALE (QS0)
24 $\overline{\text{INTA}}$ (QS1)
23 $\overline{\text{TEST}}$
22 READY
21 RESET

Figure 5.2 8088

the pin is active low. The pin names in brackets show the maximum mode functions of pins whose function is mode dependent.

5.1.1a Pins with common function in minimum/maximum mode

GROUND and Vcc
5 V power supply and ground connections.

CLK
(input)
System clock, usually provided by the special clock circuit (8284), which is designed to satisfy the close timing tolerances imposed on this input.

INTR
(input)
Maskable interrupt request input, usually from an 8259A interrupt controller.

NMI
(input)
Non-maskable interrupt. When it is used, it is usually connected to a power supply failure detection circuit or an alarm. It can also be used to interrupt an infinite program loop, when other masked interrupts cannot (see example 5.1).

RESET
(input)
Provides CPU initialisation. This signal must remain active, at high state, for at least 4 clock cycles to allow the CPU time to initialise the internal registers. This delay is provided automatically when a 8284 clock chip is used to condition the RESET signal.

Whenever the CPU is reset, the segment register CS is set to FFFFH and IP to 0. The CPU will then automatically execute the instruction held at physical addresss FFFF0H, which would normally hold a jump to the start of the main program.

RESET also resets the other segment registers and the status word to 0, which has the effect of inhibiting interrupts. This last provision is important to allow the processor to start correctly and configure the interrupt controllers before they are enabled.

READY
This input is provided to allow the CPU to access devices that cannot meet the normal CPU bus timing. As soon as a slow peripheral is addressed, it signals the

CPU via the 8284 which resets this input, synchronised to the clock.

When the device is ready, it raises the ready signal, thus allowing the CPU to continue. The CPU pauses by inserting ´WAIT´ states into its bus cycle.

$\overline{\mathbf{TEST}}$
(input)
This input also allows a further method for the CPU to be synchronised to external events.

When the CPU decodes a WAIT instruction, it tests the status of this input and only proceeds to execute the program when it is at a low level.

This input corresponds to the BUSY pin of the 8087 co-processor and provides a method for the processor to determine whether a result has been produced by the 8087.

Where there is no 8087, this input can serve to verify the end of execution of a slow operation, such as a motor start-up, read or write to a disk unit, etc.

$\overline{\mathbf{RD}}$

(tristate output)
This signal, which is active low, informs the system that the CPU is performing a read cycle. Data input occurs on the positive edge of this signal.

The output can be forced to a high impedance state to allow other masters to take control of the control bus.

AD0-AD7
(bidirectional, tristate)
This is the multiplexed low byte of the address and data buses. These pins act as outputs while the address is written and then switch to an appropriate direction for the data transfer.

The pins can be forced to a high impedance state to allow other masters to use the bus, for example, for DMA.

AD8-AD15 for the 8086
A8-A15 for the 8088
The same purpose as pins AD0-AD7, with the exception that the 8088 uses pins A8-A15 for address output only, since it only possesses an 8-bit data bus.

A19(S6)-A16(S3)
(tristate outputs)
These four pins are also multiplexed. While the address is being output, these four pins provide the top four address bits.

When the data bus is active, outputs S3 and S4 indicate which segment register was employed to generate the physical address, and consequently provide processor status information.

S4	S3	Register
0	0	ES
0	1	SS
1	0	CS (or none)
1	1	DS

At the same time line S5 provides an indication of the status of the interrupt enable flag.

Line S6 is always at 0.

These status signals are used chiefly to allow a circuit emulator to follow the operation of the CPU when a system is debugged.

$\overline{\text{BHE}}$(S7)/SSO

(tristate output, for the 8086)

While an address is being output, BHE (active low) and the address bit A0 are used to enable data on to the upper and lower halves of the data bus, allowing both 8 and 16 bit data transfers to be performed by the 8086.

During data transfers, S7 (active low) indicates whether or not the CPU is starting an interrupt acknowledge operation.

The corresponding pin on the 8088 provides the SSO status signal (in minimum mode only) and allows further status information to be obtained.

In maximum mode, this pin on the 8088 is always high and allows an 8087 to identify whether it is operating with an 8086 or with an 8088, also allowing the co-processor to adjust its data bus width accordingly.

MN ($\overline{\text{MX}}$)

This input is used to select minimum or maximum mode when tied to 5 V and ground respectively.

5.1.1b Definition of pins used in maximum mode

$\overline{\text{S2}}$, $\overline{\text{S1}}$, $\overline{\text{S0}}$

(tristate outputs)

These status lines are activated at the start of a bus cycle to allow an 8288 bus controller to determine the nature of the current bus cycle and to generate the corresponding signals.

At the end of the bus cycle all three return to a high state. As soon as one of them leaves this state it signals the start of a new operation cycle.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Operation cycle
0	0	0	Interrupt acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	HALT
1	0	0	Fetch instruction
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive state (no external access)

These three signals are forced to a high impedance state when the local bus is used by another master.

$\overline{RQ}$ ($\overline{GT0}$), $\overline{RQ}$ ($\overline{GT1}$)
(bidirectional, active low)
Each of the two pins is used to receive requests to use the local bus that are sent by other bus controllers and each is used to signal that a request will be granted at the end of the current bus cycle. The bus controller that has taken control finally informs the CPU, still via the same pin, that it can again take control of its local bus.

$\overline{RQ}$ ($\overline{GT0}$) has higher priority than $\overline{RQ}$ ($\overline{GT1}$).

This alternating input and output operation is intended for use with 8087 and 8089 co-processors and requires the designer to provide additional circuitry if a more conventional bus request protocol is needed, for example for use with DMA controllers.

$\overline{LOCK}$
(tristate output, active low)
This signal, which is asserted by writing the prefix LOCK at the start of an instruction, indicates to other bus controllers that they may not take charge of it. LOCK remains active until the instruction has been completely executed.

QS1, QS0
(outputs)
These two lines allow the 8087 numeric co-processor, or other circuits, to follow the use that the CPU makes of

its instruction queue (see below).

QS1	QS0	Operation
0	0	No operation
0	1	Fetch first byte of instruction from queue
1	0	Clear queue
1	1	Fetch next byte

This information is valid during the clock cycle that follows the operation on the instruction queue.

5.1.1c Definitions of the pins used in minimum mode

In this mode the CPU operates without a bus controller and must itself generate all the signals that are required by the system components.

M($\overline{\mathrm{I/O}}$)

(tristate output)

This signal distinguishes between a memory access and an input/output peripheral access.

It is valid throughout a complete operation cycle.

It switches to high impedance if the CPU does not have control of the bus.

The following logical inversion between the 8088 and the 8086 should be noted

	8086 M($\overline{\mathrm{I/O}}$)	8088 I/O($\overline{\mathrm{M}}$)
Memory access	1	0
I/O access	0	1

DT ($\overline{\mathrm{R}}$)

(tristate output)

This pin may be used to control the direction of the data bus buffers.

data write = 1 (transmit)
data read = 0 (receive)

As with the previous signal, it is valid throughout a complete operation cycle and switches to high impedance to free the bus if another processor requests it.

For the 8088 in minumum mode, the combination of $\overline{\mathrm{SS0}}$, DT ($\overline{\mathrm{R}}$) and I/O ($\overline{\mathrm{M}}$) allows the current cycle type to be identified

I/O($\overline{M}$)	DT($\overline{R}$)	$\overline{SS0}$	Operation
1	0	0	Interrupt acknowledge
1	0	1	Read I/O
1	1	0	Write I/O
1	1	1	Halt
0	0	0	Fetch instruction
0	0	1	Read memory
0	1	0	Write memory
0	1	1	Passive

$\overline{\text{WR}}$
(tristate output)
This active low signal indicates that the CPU is performing a write operation, and is asserted following address output.

It switches to high impedance to free the bus on request.

$\overline{\text{INTA}}$
(output) (INTerrupt Acknowledge)
This active low signal is used during interrupt acknowledgement to allow the CPU to read the interrupt number that is placed on the data bus by the interrupt controller.

ALE
(output) (Address Latch Enable)
Active high, this signal is used to latch the address at the beginning of an operation cycle.

HOLD, HLDA
(input), (output) (HOLD Acknowledge)
The active high, HOLD input is used to inform the CPU that another master wishes to use the bus.

The CPU responds by asserting the active high HLDA signal at the end of the current operation cycle. At the same time, it puts its own bus control signals into a high impedance state .

When the local bus controller has completed its use of the buses, it lowers the HOLD request signal. The CPU responds by lowering HLDA and resuming bus control.

$\overline{\text{DEN}}$
(tristate output) (Data ENable)
Active low, DEN is used to enable all data transfers, and consequently can be used, with other logic, to control isolation of the local bus when another processor takes control of the system, or during a direct memory access.

5.1.2 Internal organisation

The CPU consists of two distinct units.

Execution Unit (EU)

In essence this part contains the arithmetic and logic unit (ALU) and executes all the instructions. The ALU contains the CPU status and control words and manipulates the operands and general registers. All the registers and internal buses are 16 bits wide.

The EU is isolated from the external environment. It receives instructions and operands from an internal memory, an instruction queue that functions as a First-In-First-Out (FIFO) buffer, controlled by the interface unit.

This queue is 4 bytes long for the 8088 and 6 bytes long for the 8086.

When an instruction requires an operand to be fetched from memory, the EU requests the fetch from the interface unit.

Bus Interface Unit (BIU)

This part carries out all bus operations on behalf of the EU. It contains segment registers and it calculates the 20-bit physical address. It also looks after address management for the 8087 numeric co-processsor (see also chapter 7).

The BIU also contains the instruction pointer register (IP) and uses its contents to fetch instructions from memory depending upon the availability of the buses and the instruction queue status.

This operation is called prefetching and helps to improve performance considerably. Generally, the EU will almost always have available to it, stored in the BIU, the next following bytes of code.

The BIU prefetches instructions based on simple incrementing of the IP; consequently, a branch (either a jump or subroutine call) to an instruction outside the scope of the instruction queue necessitates the queue being cleared. When this happens, the EU inevitably slows until the queue can be replenished.

This feature can be useful in optimising the performance of a loop, since tests chosen to allow the most common results to cause execution to continue at the next address, rather than requiring a branch, will execute faster.

5.1.2a Prefetch

The following example illustrates the time advantage gained as a result of the separation of the EU and BIU tasks and from the use of the instruction queue.

Thus the program

```
(1) MOV [BX],AL   ; write data
(2) MUL BX        ; instruction without
                    external access
(3) MOV CX,2      ; instruction with external
                    operand read
(4) ADD SI,CX     ; instruction without
                    external access
(5) ADD DI,CX     ; instruction without
                    external access
```

would execute without prefetch in the following way

```
CPU:
|Exec 1|  |Wri 1||Fetch 2|  |Exec 2|  |Fetch 3||Rd Op 3|  |Exec 3|

Bus occupation
          xxxxx xxxxxxx           xxxxxxx xxxxxxx
```

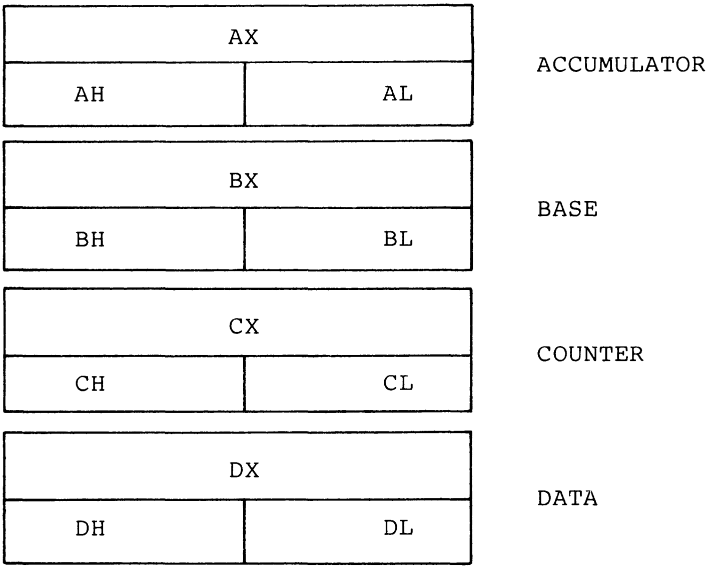
While with prefetch, bus operation is optimised.

```
EU:      |Execute 1|         |Execute 2|         |Execute 3|

BIU:|Fetch 2||Fetch 3| |Write 1||Rd Op 3| |Fetch 4||Fetch 5|

Bus occupation
     xxxxxxx xxxxxxx  xxxxxxx xxxxxxx  xxxxxxx xxxxxxx
```

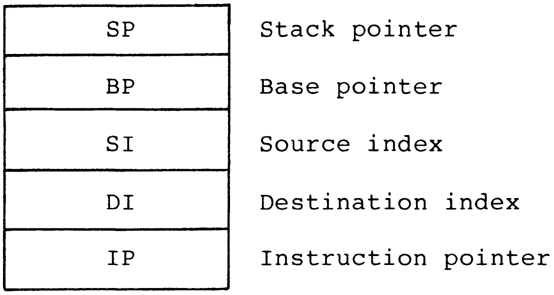
5.1.2b Registers

The set of registers accessible to the programmer can be divided into the four groups, described in chapter 1.

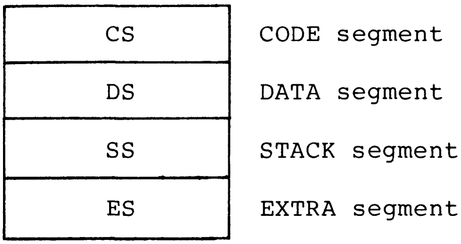
Data group


AX
AH
AL
ACCUMULATOR
BX
BH
BL
BASE
CX
CH
CL
COUNTER
DX
DH
DL
DATA

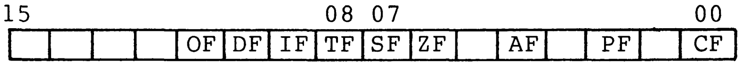
Pointer and index group


SP
Stack pointer
BP
Base pointer
SI
Source index
DI
Destination index
IP
Instruction pointer

Segment group


CS
CODE segment
DS
DATA segment
SS
STACK segment
ES
EXTRA segment

Status word (Flags)


15
08
07
00
OF
DF
IF
TF
SF
ZF
AF
PF
CF

Six bits reflect the results of an arithmetic or logical operation and three take part in the control of the processor.

The first six are as follows.

AF

The auxiliary flag indicates a carry between 4 bit groups of a 8 bit quantity. It is used implicitly for arithmetic decimal matching. In fact, the sum of two decimal numbers that are considered by the CPU like hexadecimal numbers, gives a hexadecimal number

3D + 5D = 8H (AH = 0)
4D + 9D = 0DH (AH = 1) <- (overflow of 9)

In the case of an operation beyond decimal matching the CPU adds 06 if the flag AF has been set

AF = 0 -> 8H -> 8D
AF = 1 -> 0DH + 06 -> 13D

CF

The carry flag indicates that the result of an operation has caused an overflow of the most significant digit of the byte or word specified in the instruction.

This flag is used implicitly, as a carry, for concatenated arithmetic calculations, or explicitly for certain conditional jumps.

OF

The overflow flag indicates that an operation between bytes or between words has exceeded by more than 1 the capacity of a byte or of a word respectively. The result can no longer be represented, even with carry. This error causes a specific interrupt with the help of the instruction INTO.

SF

The sign flag reproduces the value of the most significant bit of a signed 8 or 16 bit quantity.

The signed arithmetic operates in two´s complement and negative numbers therefore have to have the most significant bit set.

SF = 0 positive number
SF = 1 negative number

This flag is used explicitly for certain conditional jumps.

ZF
The zero flag indicates that the result of an arithmetic or logical operation is zero. It is used in several conditional jump instructions.

PF
The parity flag indicates that the number of bits having the value 1, in a byte or a word, is an even number. This flag is used with certain jumps, in particular to test the integrity of data in communications software.

DF
The direction flag fixes the direction of repetitive operations (string instructions).

DF = 0 -> index increment
DF = 1 -> index decrement

IF
The interrupt enable flag indicates whether or not interrupts are currently enabled.

IF = 1 -> interrupts allowed
IF = 0 -> interrupt requests not acknowledged

TF
The trap flag puts the CPU in single step mode to aid execution fault finding.

5.2 Bus Transfers

5.2.1 Read and write cycle

The CPU communicates with its environment by means of a 20-bit bus time-multiplexed into the address, data and status buses, and by means of a control bus.

In order to carry out a data transfer or read an instruction, the CPU executes an operation cycle, counting at least four clock cycles, known as T1, T2, T3 and T4, and shown in figures 5.3, 5.4, 5.5 and 5.6.

The 20-bit address is output during T1. The trailing edge of the ALE signal is timed to allow it to be used to latch the address so that it may be maintained throughout the complete operation cycle.

The data transfer (8 or 16 bits) takes place during T3 and T4.

The read ($\overline{RD}$) or write ($\overline{WR}$) signals are always active during T2.

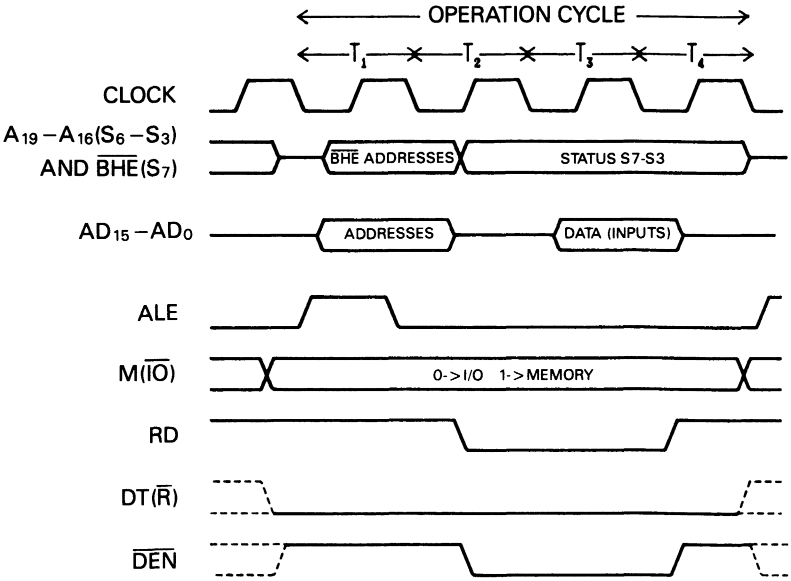

OPERATION CYCLE
T1
T2
T3
T4
CLOCK
A19 –A16(S6 –S3) AND BHE(S7)
BHE ADDRESSES
STATUS S7-S3
AD15 –AD0
ADDRESSES
DATA (INPUTS)
ALE
M(IO)
0->I/O 1->MEMORY
RD
DT(R)
DEN

Figure 5.3

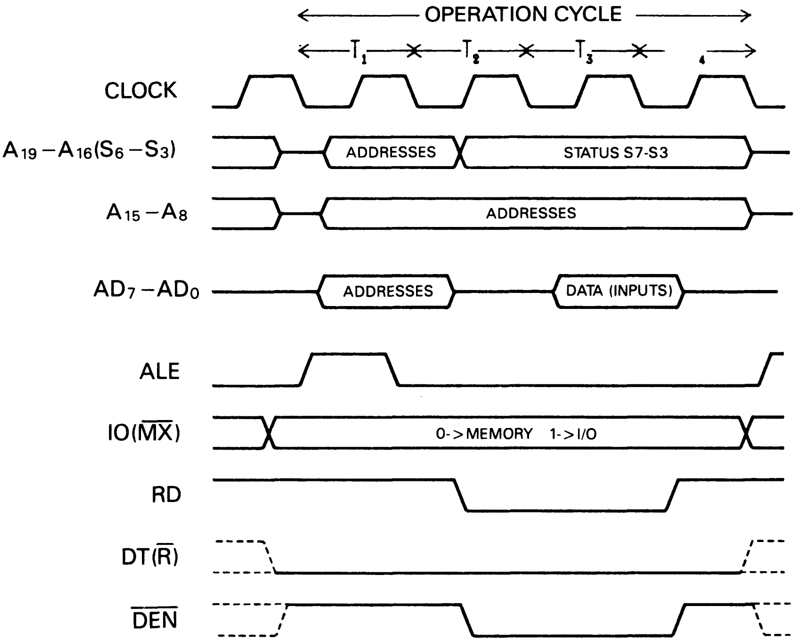

OPERATION CYCLE
T1
T2
T3
4
CLOCK
A19 –A16(S6 –S3)
ADDRESSES
STATUS S7-S3
A15 –A8
ADDRESSES
AD7 –AD0
ADDRESSES
DATA (INPUTS)
ALE
IO(MX)
0->MEMORY 1->I/O
RD
DT(R)
DEN

Figure 5.4

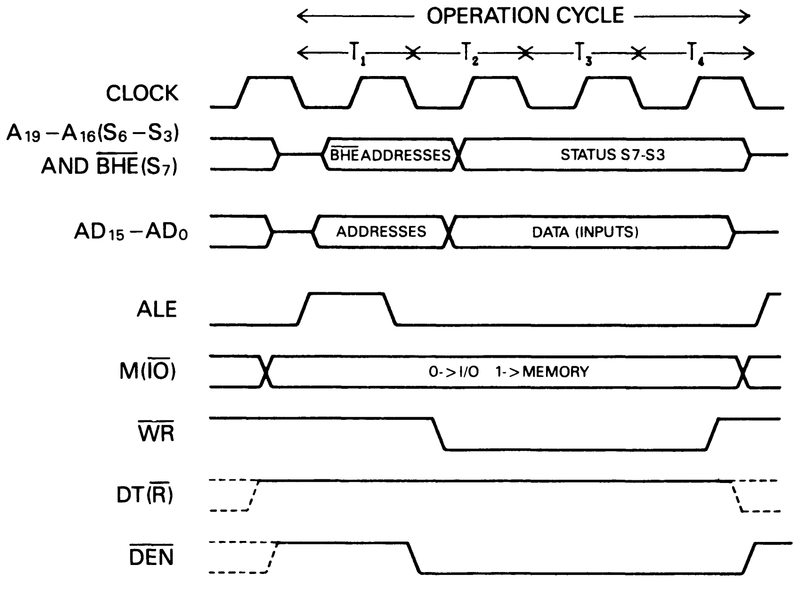

OPERATION CYCLE
T1
T2
T3
T4
CLOCK
A19 – A16(S6 – S3) AND BHE(S7)
BHE ADDRESSES
STATUS S7-S3
AD15 – AD0
ADDRESSES
DATA (INPUTS)
ALE
M(IO)
0->I/O 1->MEMORY
WR
DT(R)
DEN

Figure 5.5

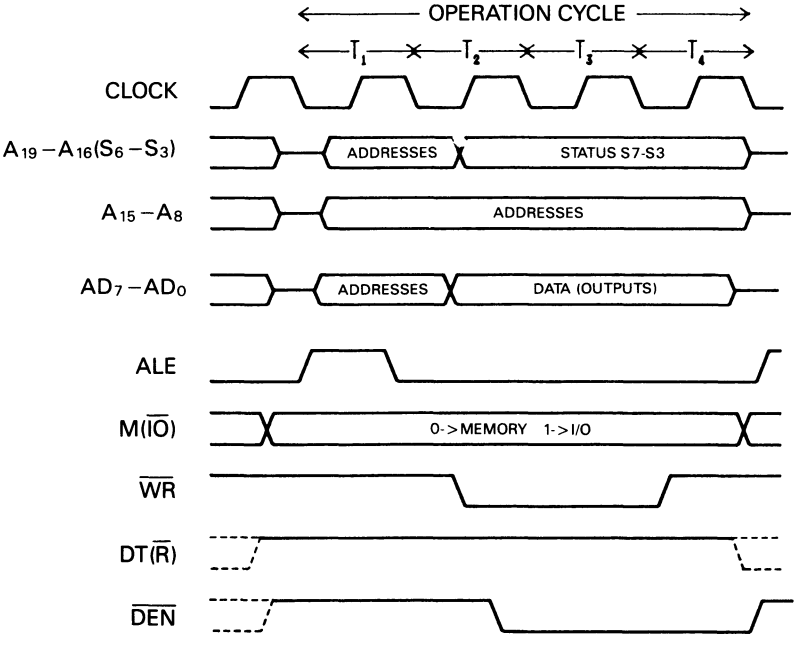

OPERATION CYCLE
T1
T2
T3
T4
CLOCK
A19 – A16(S6 – S3)
ADDRESSES
STATUS S7-S3
A15 – A8
ADDRESSES
AD7 – AD0
ADDRESSES
DATA (OUTPUTS)
ALE
M(IO)
0->MEMORY 1->I/O
WR
DT(R)
DEN

Figure 5.6

If the selected memory or peripheral is not in a position to complete the transfer with the normal CPU timing, the peripheral must inform the CPU, usually via the 8284 clock circuit, using the processor´s READY input.

This change of state must occur before T3 if the CPU is to be able to insert wait states (TW) after T3.

As soon as the selected circuit is ready, it returns READY to its normal state, via the 8284, and the CPU completes its bus cycle.

Status bits S3 to S7 for the 8086 and S3 to S6 for the 8088 may be demultiplexed from the high address bits (and from the BHE bit for the 8086) during cycles T2, T3 and T4.

Finally, passive states are inserted when the instruction queue has been completely filled by the BIU. These states are detectable by means of the status lines.

5.2.2 Organisation of memory space for the 8086

The 8086 must be able to read or write either a byte or a word. In the case of a byte it may be found at either an even or an odd address. Similarly, a word may begin at an even or odd address.

In order to solve this addressing problem, the memory space has to be divided into two byte-wide blocks of up to 512K each. One block is connected to the lower half of the data bus (D0-D7) and responds to even addresses (A0 = 0). The other block is connected to the upper half of the bus (D8-D15) and responds to odd addresses (A0 = 1).

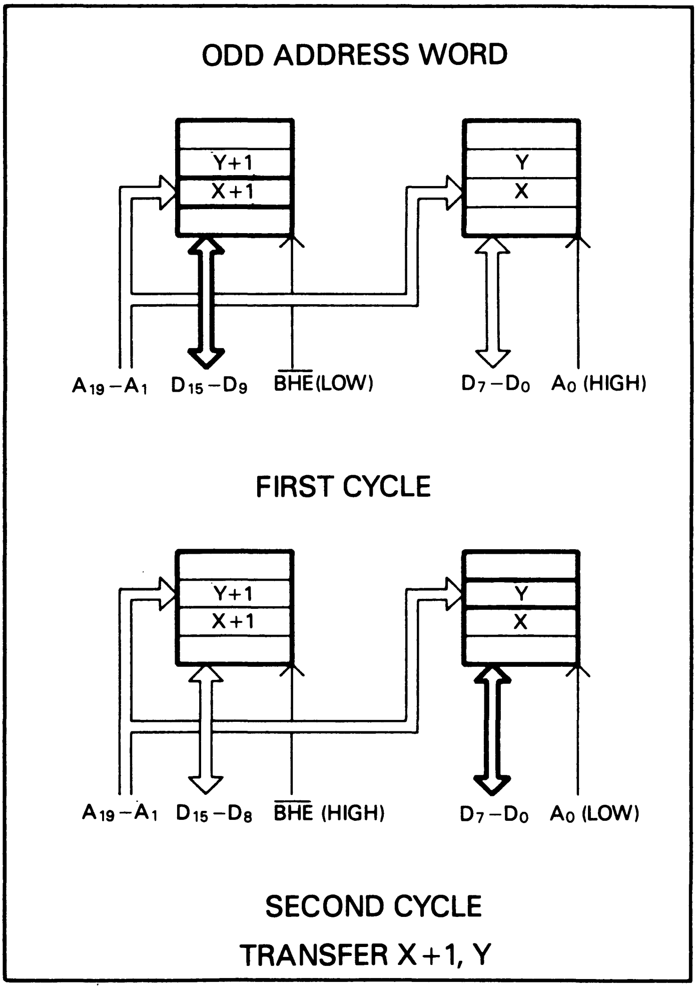
If a 16-bit word begins at an odd address, it will always be read in two passes: one byte at the odd address, then one byte at the even address immediately above.

However, a 16-bit word located at an even address may be read in a single bus cycle with the bytes being accessed in both odd and even memory banks concurrently.

It is therefore necessary to be able to determine, for each address output, whether a word or byte transfer is to be performed and, in the latter case, whether the odd or even address byte is being accessed. This information may be obtained from the BHE (Bus High Enable) signal and A0 address bit in accordance with the following table. The BHE signal may be latched using ALE.

$\overline{\text{BHE}}$	A0	Write or Read
0	0	Word
0	1	Byte - odd address
1	0	Byte - even address
1	1	No operation

ODD ADDRESS WORD


Y+1
X+1
Y
X
$A_{19}-A_1$
$D_{15}-D_9$
$\overline{\text{BHE}}$(LOW)
D_7-D_0
A_0 (HIGH)
Y+1
X+1
Y
X
$A_{19}-A_1$
$D_{15}-D_8$
$\overline{\text{BHE}}$ (HIGH)
D_7-D_0
A_0 (LOW)

FIRST CYCLE

SECOND CYCLE

TRANSFER X+1, Y

Figure 5.7

Address bits A19-A1 select a byte in each of the two blocks. $\overline{\text{BHE}}$ and A0 choose one or the other, or both.

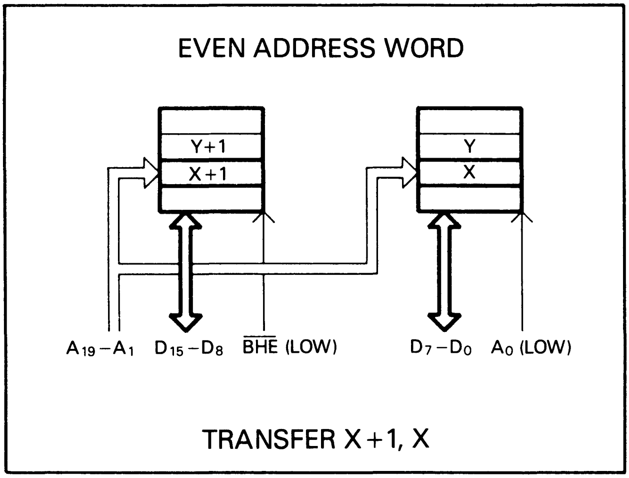

EVEN ADDRESS WORD
Y+1
X+1
Y
X
$A_{19}-A_1$
$D_{15}-D_8$
$\overline{BHE}$ (LOW)
D_7-D_0
A_0 (LOW)
TRANSFER X+1, X

Figure 5.8

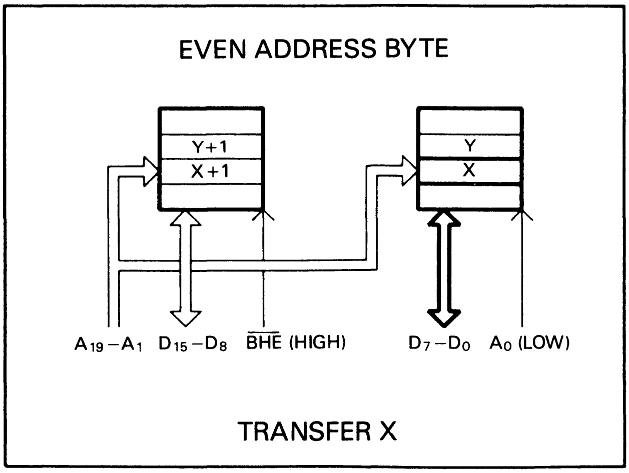

EVEN ADDRESS BYTE
Y+1
X+1
Y
X
$A_{19}-A_1$
$D_{15}-D_8$
$\overline{BHE}$ (HIGH)
D_7-D_0
A_0 (LOW)
TRANSFER X

Figure 5.9

It should be noted that the CPU does not necessarily write D0-D7 to the lower byte of a 16-bit register (for example, AL) and D8-D15 to the upper byte.

In the case of an operation on a word located at an even address, the first cycle will read the low part of the word on D8-D15 and the second, the high part on D0-D7.

Although a 16-bit word is referenced by the programmer, independently of whether or not it is situated at an even address, it is important to note that word transfers located at an odd address extends the execution time. In the case of an array of words, the penalty may become significant. For this reason the EVEN assembler directive is provided to reserve memory storage on even address boundaries.

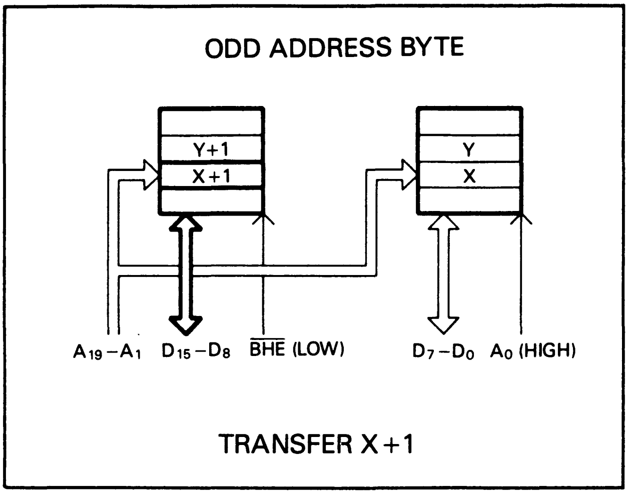


Figure 5.10

It is important to appreciate that these remarks do not apply to the 8088 which necessarily operates in bytes and does not require any $\overline{\text{BHE}}$ signal.

5.3 Circuits in the 8086/8088 family

5.3.1 8284A clock generator

The main function of this circuit is to provide the clock signals, not only for the CPU but also for other peripheral circuits such as timers and serial transmitters (USART). It is also responsible for initialising the CPU and peripherals via the RESET signal and for synchronising the processor by controlling the READY pin.

The actual clock signal is provided either by an external clock or a crystal, and is determined by the state of the $F(\overline{C})$ pin.

When $F(\overline{C})$ is connected to ground, the frequency source is controlled by the quartz crystal (inputs X1, X2), whose resonance frequency must be three times the required processor clock rate.

When $F(\overline{C})$ is connected to the 5 V power supply, the source is a TTL compatible signal, connected to the EFI

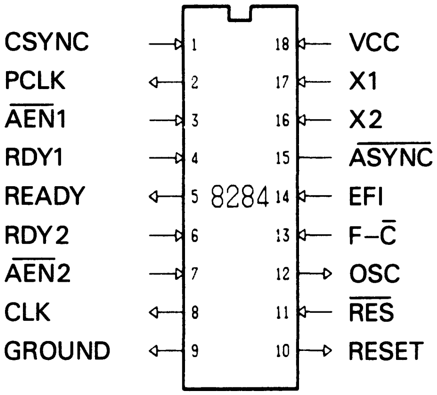


Figure 5.11 8284A clock generator
(NC = not connected or ASYNC for the 8284A)

pin, also with a frequency three times that of the CPU.

The OSC oscillator output of the 8284 delivers a frequency equal to that of the crystal. It is intended to be connected to the EFI inputs of other 8284A circuits in multiprocessor systems. In such systems each processor must have its own clock generator because of the necessity of independent control of each processor READY signal.

The CLK clock output is the clock for the CPU and coprocessors working on the same local bus.

The peripheral clock PCLK produces a clock signal at half the frequency of CLK. It is usually used by peripherals.

In addition, CLK and PCLK can be synchronised to an external event. When the CSYNC input is taken high, it forces the CLK and PCLK outputs high. When CSYNC returns low, the clock signals resume on the trailing edge of the frequency source.

The $\overline{\text{RES}}$ input controls the CPU's RESET input. The 8284A RESET output is synchronised with the clock and can be used as a general RESET for the system.

The READY input may be driven by those circuits that are unable to transfer data at the CPU bus frequency. It is also used in multiprocessor systems to force the CPU to wait before accessing the MULTIBUS.

The following are the two possible approaches.

1. The system is normally NOT READY. When the selected circuit has had time to execute the transfer, it asserts the CPU READY signal.

This is the method adopted in large systems.

If the READY signal does not arrive in time, unnecessary wait cycles (TW) are inserted.

2. The second technique is to assume that all the circuits are fast enough. If the chosen circuit is slow, it must deassert READY. This technique is used in single CPU systems. If, however, the signal is not deactivated in time, the cycle terminates prematurely and the data transferred is likely to be corrupted.

The 8284A has two inputs RDY 1 and RDY 2 and a single synchronised READY output. The inputs $\overline{\text{AEN1}}$ and $\overline{\text{AEN2}}$, which are active low, allow the inputs RDY1 and RDY2 to be selected.

5.3.2 8282/8283 8-bit address latch

The main function of this circuit is to latch the address using the ALE strobe signal connected to the STB (strobe) pin.

ALE is provided by the CPU in minimum mode or by the 8288 bus controller in maximum mode.

The latched address remains present throughout the entire operation cycle even if the cycle includes state waits (TW).

This device also has to provide sufficient drive to supply circuits connected to the address bus.

Another facility provided is to isolate the local bus by switching all the outputs to a high impedance state upon request from another system bus master, or in the event of a direct memory access.

This function is carried out by the input $\overline{\text{OE}}$ (output enable, active low). In systems having only a single processor this input may be hardwired to ground.

The 8283 has inverted outputs, whereas the 8282 does not.

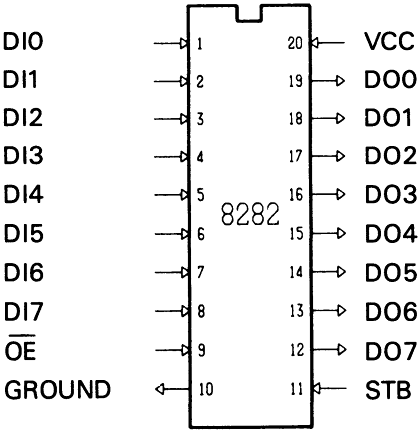


Figure 5.12 8282 latch
(In the 8283 the outputs are inverted.)

5.3.3 8286/8287 8-bit data transceivers

The main function of this circuit is to buffer the data bus lines to provide sufficient drive power for the system.

It is not necessary to latch data since the CPU maintains the data valid for write operations throughout the entire transfer.

Control of the direction of data flow through the bidirectional transceiver for read and write operations is obtained by connecting the DT(R), data transmit - data receive output, that comes from the CPU in minimum mode or from the 8288 in maximum mode, to pin T of the 8286/8287. If T is high, the data present at A0-A7 is output on pins B0-B7 when OE is asserted.

The OE signals may be controlled either by inverting the DEN output of the 8288 for a maximum mode CPU, or directly at the DEN output of the CPU in mimimum mode.

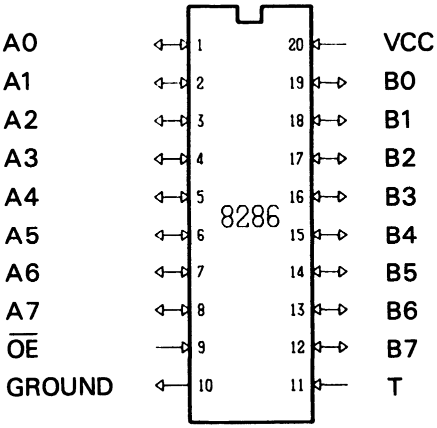


Figure 5.13 8286 transmitter
(In the 8287 the outputs are inverted.)

5.3.4 8288 bus controller

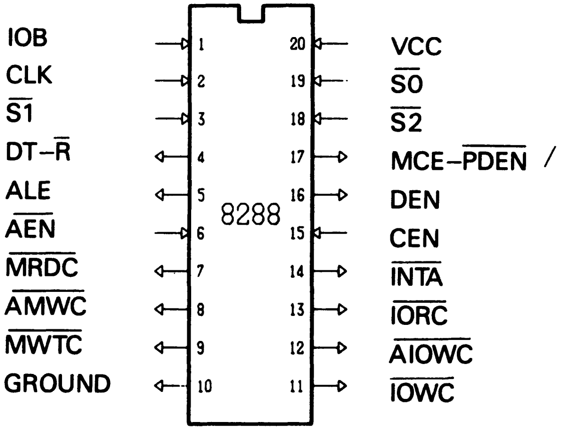

IOB
CLK
$\overline{S1}$
DT-$\overline{R}$
ALE
$\overline{AEN}$
$\overline{MRDC}$
$\overline{AMWC}$
$\overline{MWTC}$
GROUND
8288
VCC
$\overline{S0}$
$\overline{S2}$
MCE-$\overline{PDEN}$ /
DEN
CEN
$\overline{INTA}$
$\overline{IORC}$
$\overline{AIOWC}$
$\overline{IOWC}$
1
2
3
4
5
6
7
8
9
10
20
19
18
17
16
15
14
13
12
11

Figure 5.14 8288 pin assignment

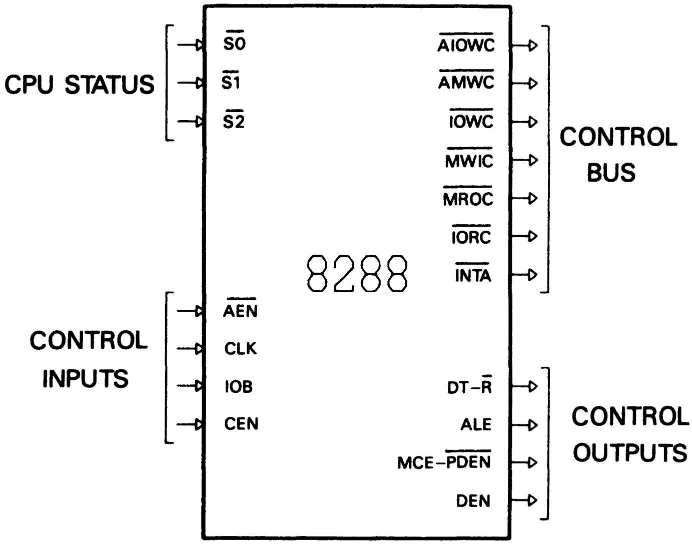

CPU STATUS
$\overline{S0}$
$\overline{S1}$
$\overline{S2}$
CONTROL INPUTS
$\overline{AEN}$
CLK
IOB
CEN
8288
$\overline{AIOWC}$
$\overline{AMWC}$
$\overline{IOWC}$
$\overline{MWIC}$
$\overline{MROC}$
$\overline{IORC}$
$\overline{INTA}$
CONTROL BUS
DT-$\overline{R}$
ALE
MCE-$\overline{PDEN}$
DEN
CONTROL OUTPUTS

Figure 5.15 8288 functions

The controller decodes the three status lines of a maximum mode CPU - S0, S1 and S2 - in order to generate the following bus control signals.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	CPU activity	8288 command
0	0	0	Interrupt acknowledge	$\overline{\text{INTA}}$
0	0	1	Read I/O peripheral	$\overline{\text{IORC}}$
0	1	0	Write I/O peripheral	$\overline{\text{IOWC}}$, $\overline{\text{AIOWC}}$
0	1	1	Halt	None
1	0	0	Read code	$\overline{\text{MRDC}}$
1	0	1	Read memory	$\overline{\text{MRDC}}$
1	1	0	Write memory	$\overline{\text{MWTC}}$, $\overline{\text{AMWC}}$
1	1	1	Passive	None

Unlike a minimum mode CPU, the 8288 splits read operations, controlled by the RD signal in minimum mode systems, into separate memory (MRDC) and I/O (IORC) read signals to distinguish I/O references from memory references.

The same applies for writing; WR becomes IOWC for the I/O field and MWTC for the memory field. It should also be noted that for writing two new signals AIOWC and AMWC are created. The A prefix denotes that the strobe occurs in advance of its normal timing. This allows more time for peripherals or slower memories to react, and reduces the need for state waits (TW).

One further input, AEN, is provided to enable the control signal outputs. The function of this input is dependent upon the mode of operation of the 8288.

The following sections describe the two possible modes of operation that are selectable using the IOB pin.

I/O bus mode

(IOB high)

This configuration is suited to the case of a multiprocessor system where the CPU that uses the 8288 has its own I/O peripheral bus. This arrangement allows a CPU to have some local resources that may be accessed without involving the remainder of the system. Consequently, accesses to resources on the I/O bus may proceed independently of the system resource arbitration procedure.

Arbitration of requests for system resources can only proceed after a stable address is available. To allow for this delay the 8288 delays some of its control signals by 105 ns.

In the I/O bus mode, this delay is not introduced for IORC, IOWC, AIOWC and INTA, which are also always enabled, since the AEN input (Address Enable) has no effect on them.

When set at logic 1, the IOB input also has the effect of selecting function PDEN on pin 17. This function is equivalent to DEN but is especially for peripherals and combines with DT-R to control the I/O bus transceivers.

System bus mode

(IOB low)

This mode is used where a multi-master system shares resources in both the memory field and in the I/O field.

The set of 8288 control signals is put under the control of the AEN input and none of them is activated until 105 ns have elapsed.

Arbitration is therefore possible for both fields.

When AEN is not active all the control outputs remain in a high impedance state.

In this mode, pin 17 provides the Master Cascade Enable (MCE) function which allows a slave interrupt controller on the system bus to be addressed during an interrupt acknowledge (see section 5.4).

For both modes, Command Enable (CEN) inhibits all DEN and PDEN commands when it is selected. This input is used to regulate the contention for access to circuits available on the system bus with those available on the resident bus, in cases where the latter might be located at the same address.

5.4 Interrupts

The CPU can process up to 256 different interrupt types, numbered from 0 to 255.

These interrupts can be generated by program instructions. In addition, the CPU can itself generate interrupts in cases of overflow, division by zero or TRAP.

5.4.1 External interrupts

The CPU has two pins (NMI and INTR) on which external interrupts may be signalled.

NMI is a type 2 non-maskable external interrupt; it is used to counter a catastrophic event or in order to extricate the processor from the execution of an infinite loop where no other interrupt is possible.

INTR is generated by an interrupt controller (8259A), which is itself connected to the circuits that need to interrupt the CPU. One or more of these circuits can themselves be interrupt controllers; they are then called 'slaves' of the 'master' circuit connected to the CPU.

The 8259A is a programmable circuit (see section 5.4.5) capable of selectively receiving interrupt requests, resolving priority problems, activating the CPU´s INTR pin and providing the CPU with the stored interrupt type.

When INTR is active, the state of the IF flag of the status word conditions the response of the CPU. At best, the interrupt will only be taken into account at the end of the complete execution of the current instruction, including any prefix, such as REP, LOCK or segment corrector.

It should be noted that in the case of an instruction modifying a segment register, such as POP DS or MOV SS, AX, the interrupt will only be taken into account at the end of the following instruction, even in the case of a non-maskable NMI interrupt.

Thus, in the case of a stack change the following sequence

```
   MOV AX, NEW-STACK
(a)
   MOV SS, AX
   MOV SP, OFFSET NEW-TOP
(b)
```

cannot be interrupted between (a) and (b). This arrangement thus avoids the risk of using an incompletely defined stack. It is therefore important for the loading of the SP register to be carried out immediately after the loading of SS.

On the other hand, a repeated loop instruction can be interrupted at the end of a sequence but before its complete execution.

Similarly, execution of a WAIT instruction, which tests the state of the TEST pin, can be suspended.

If IF = 0, the interrupt is masked and the CPU continues execution of the program so long as IF remains at this state. This masking is initiated by the CLI instruction (Clear Interrupt enable flag).

If IF = 1, the interrupt is allowed and the CPU executes the switching sequence. The STI (Set Interrupt enable flag) instruction sets IF to 1.

The masking of interrupts can also be done within the 8259A itself. In this case, even if IF = 1, it will also be necessary for the interrupt type to be permitted by an appropriate interrupt mask currently being active in the interrupt controller. Thus, the inhibition or selective authorisation of a particular external interrupt is possible.

The CPU acknowledges the interrupt by carrying out, in conjuction with the bus controller, two acknowledge cycles.

These cycles behave like I/O reads, but the $\overline{\text{INTA}}$

signal is generated in place of $\overline{RD}$ with the same duration and timing characteristics in the operation cycle.

The first cycle warns the 8259A that the interrupt is being granted. The second allows the 8259A to provide the CPU via the data bus with the type of interrupt that it has accepted.

Provided with this information, the CPU can then execute the corresponding interrupt procedure.

5.4.2 Internal interrupts

An internal interrupt is initiated by the INT instruction. The interrupt type is placed in the INT operand and tells the CPU which procedure to execute.

An internal interrupt can also be initiated automatically in the following three cases.

1. Flag OF = 1 indicates an overflow. Interrupt type 4 is generated by the special instruction INTO.

2. The result of a division is of greater size than its destination (8 or 16 bits). This is a case of division by zero and a type 0 interrupt is generated.

3. Flag TF has been set to 1 (TRAP); the CPU then generates a type 1 interrupt after each instruction. This arragement allows the single step testing of a program. The type 1 interrupt can for example load the contents of the CPU registers into a special memory zone that can be read or modified. The interrupt procedure then reloads the registers and returns to the main program.

All the internal interrupts have the following characteristics:

there is no INTA operation cycle
they are not maskable, except for single step
the order of priority is as follows

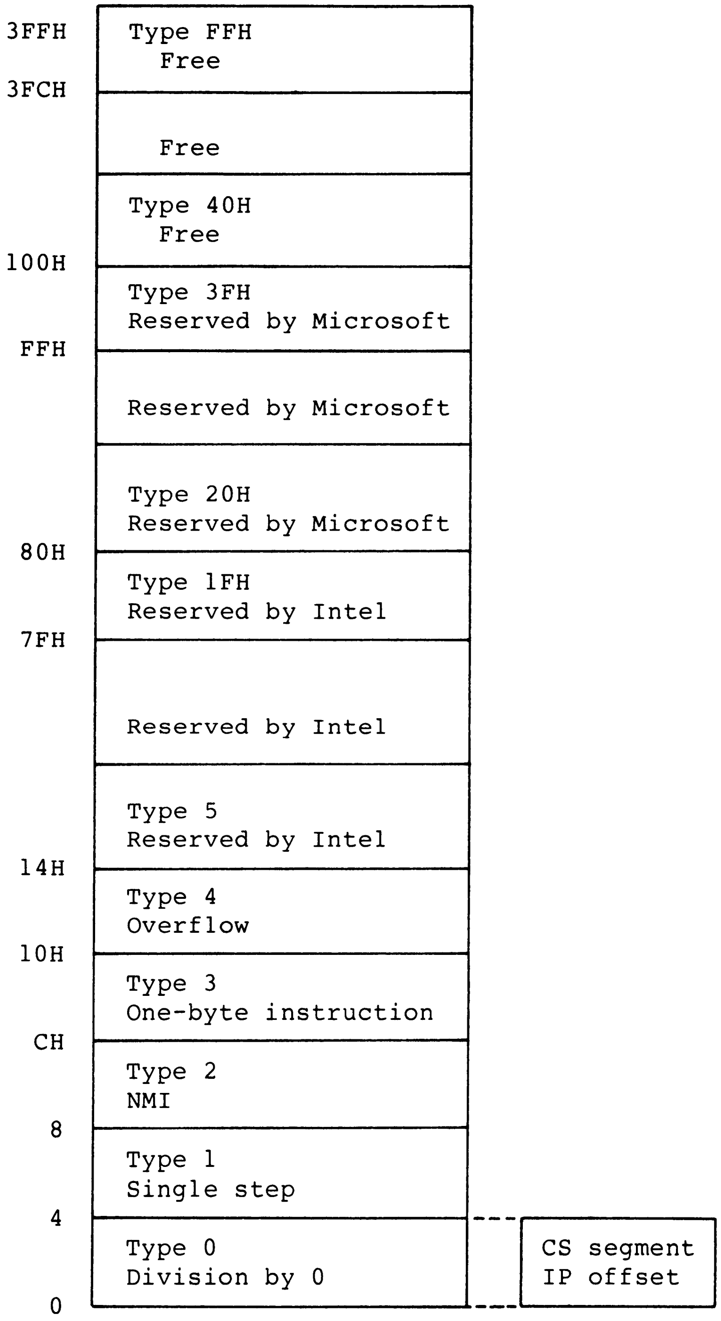
1. Division by zero, INT_n, INTO (the first encountered)
2. NMI
3. INTR (external)
4. Single step

5.4.3 Interrupt pointer table

This table must be located in the first 1024 bytes of memory (from 0 to 3FFH).

It is made up of 256 pointers (maximum) containing the addresses of the service routines corresponding to each interrupt type.

A particular pointer´s address may be calculated by multiplying the interrupt type by four. The pointer is made up of two 16-bit words. The first, at the lower address, is the IP offset, and the second, higher address word is the CS value.


3FFH
Type FFH
Free
3FCH
Free
Type 40H
Free
100H
Type 3FH
Reserved by Microsoft
FFH
Reserved by Microsoft
Type 20H
Reserved by Microsoft
80H
Type 1FH
Reserved by Intel
7FH
Reserved by Intel
Type 5
Reserved by Intel
14H
Type 4
Overflow
10H
Type 3
One-byte instruction
CH
Type 2
NMI
8
Type 1
Single step
4
Type 0
Division by 0
0
CS segment
IP offset

The table is normally placed in RAM and initialising it is one of the first tasks that must be performed once a processor is started, and certainly before interrupts are enabled.

The location of the interrupt service routine pointers in RAM allows a program to modify routines if necessary. However, care should be taken in the choice of interrupt types since some are reserved for proprietary software.

Thus, for example, Microsoft´s MSDOS reserves interrupts of type 20H to 3FH, while Intel uses interrupts of type 0 to 1FH.

5.4.4 Branch to interrupt routine

After reading the interrupt type and calculating the corresponding address of the pointer in the table, the CPU carries out the following operations:

- the status word is written on to the stack;
- the contents of CS are written on to the stack and replaced with the upper word of the service routine pointer;
- the contents of IP are written on to the stack and replaced with the lower word of the service routine pointer;
- the TF and IF flags are set to zero, thus inhibiting further interrupts and single-step operation. Further interrupts are suspended to allow the programmer to reappraise interrupt priorities in the light of the interrupt currently being serviced.

Return is effected with the aid of the IRET instruction and not RET. This might otherwise be a possible source of error because in an ordinary subroutine the status word is not saved. Any confusion would result in a serious fault in the functioning of the stack.

IP and then CS are reloaded and the status word restored last; the CPU too is restored to the same condition as it was before the interrupt.

At the end of an interrupt service routine, it is important to establish that the interrupt request which caused the service routine to be has been removed. If this were not the case, the routine would be executed again with possibly undefined results. Furthermore, if the 8259A is operating in the specific acknowledge mode, one must not forget to send the interrupt end control word.

5.4.5 8259A PIC - Programmable Interrupt Controller

Pin assignment

D7-D0	8-bit bidirectional data bus
RD	read command (connected to IORDC on the 8288)
WR	write command (connected to AIOWC on the 8288)
A0	select command (connected to A1 on the 8086 and to A0 on the 8088)
CS	chip select from address decoder circuits
CAS2-CAS0	cascade lines
SP or EN	slave activation or buffer control
INT	interrupt request (connected to INTR on CPU)
INTA	interrupt acknowledge (connected to INTA on the 8288)
IR1-IR7	interrupt request inputs

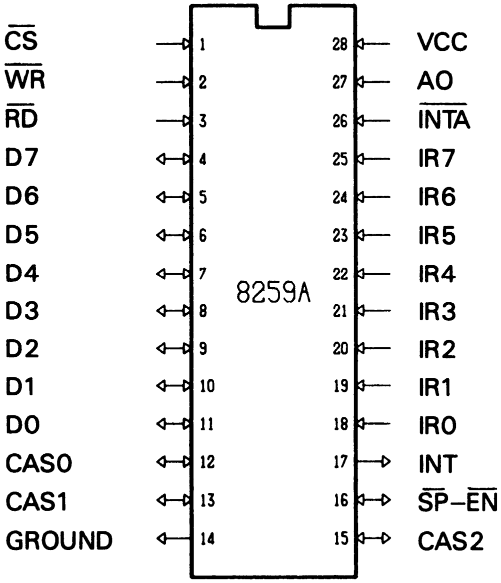

CS 1
WR 2
RD 3
D7 4
D6 5
D5 6
D4 7
D3 8
D2 9
D1 10
D0 11
CAS0 12
CAS1 13
GROUND 14
8259A
28 VCC
27 A0
26 INTA
25 IR7
24 IR6
23 IR5
22 IR4
21 IR3
20 IR2
19 IR1
18 IR0
17 INT
16 SP-EN
15 CAS2

Figure 5.16 8259A Programmable Interrupt Controller

Method of operation

The 8259A receives eight interrupt requests (IR0-IR7) and detects either a change of state on the pin or a high level. The Interrupt Request Register (IRR) stores this request according to its rank.

According to the chosen priority type the 8259A verifies that the last received request is not masked and that it is of higher priority than the one currently being serviced. If so, the CPU´s INTR pin is activated.

If the interrupts are enabled in the CPU, the latter initiates an acknowledge cycle by generating INTA.

At this point the bit from IRR that initiated the interrupt is written into the 8-bit In Service Register ISR. The corresponding bit is reset to 0 in IRR.

When the second $\overline{\text{INTA}}$ is received, the 8259A provides an 8-bit interrupt type by taking the first 5 bits from a register loaded on initialisation and completing the remaining 3 bits according to the rank of the current interrupt request in the order IR0-IR7.

There are then two possibilities

1. Automatic interrupt mode has be chosen, in which case the appropriate bit in ISR is reset to zero at the end of the second INTA pulse. The interrupt is said to be serviced as soon as the interrupt type has been read.
2. Specific interrupt end mode has been chosen. In this case, the programmer must provide for a command word to be sent telling the 8259A to consider this interrupt to be serviced and to delete the appropriate bit in ISR.

Clearly, in the second method if the interrupt is not cleared before the service routine is exited, the corresponding interrupt will always be considered as being serviced and will no longer be able to interrupt the CPU.

Despite this risk of being forgotten, this is the safer method of operation because it is often the execution of the interrupt service routine that causes the interrupt request to disappear. If the automatic mode has been chosen and interrupts have been enabled in the service routine before the request is removed, there is the danger that the program may jump back on itself and completely fill the stack.

Additionally, this mode allows the CPU to know if other interrupts have themselves been interrupted by re-reading the ISR register of the 8259A. Other bits set at 1 in this register will reveal those service routines that are waiting. By this means, the programmer can prevent one or more interrupt service routines from being held up by another interrupt service routine.

Cascade connection

On initialisation, the master is informed that one or more IR pins support a slave 8259A. When this pin is selected to interrupt the CPU, the master places the pin´s rank on the cascade output lines. The appropriate slave ´knows´ that it is involved because on its own initialisation it was informed which priority input it is connected to, and consequently it can recognise its address on the cascade outputs.

This exchange is made when the first $\overline{\text{INTA}}$ is received, whether by the master or by the one or more of the slaves.

Subsequently, two EOI (End Of Interrupt) commands are required, one for the master and the other for the slave.

Programming

The 8269A accepts two kinds of byte formatted command.

- Initialisation commands made by a sequence of 3 writes if there is only one 8259A, or 4 writes if 8259A slaves are also involved.
- Operating commands generated at any time during the program.

In the following command formats it should be noted that X indicates that the value of the bit can be placed at 0 or 1.

Initialisation

ICW (Initialisation Command Word)

The initialisation bytes must be sent in the following order

ICW 1

A0	7							0
0	X	X	X	1	LTIM	1	SNGL	1

LTIM (Level Triggered Input Mode):

1 indicates detection of a level on IR_i

0 indicates detection of a state change (leading edge) on IR_i

SNGL (Single):

1 indicates to the 8259A that it is alone

0 indicates that the 8259A is not alone. ICW 3 is therefore necessary.

ICW 2

A0	7							0
1	T7	T6	T5	T4	T3	X	X	X

T7-T3: These bits are used to generate the interrupt types. T7-T3 are concatenated with the three bits generated automatically according to the order number of the IR pin that initiated the interrupt to form the 8-bit interrupt type that will be signalled to the CPU.

ICW 3
(Only required if SNGL of ICW 1 is at 0)

This word is interpreted by the 8259A in two different ways depending on whether a master or a slave is involved. The correct interpretation is achieved either by hardwiring the $\overline{\text{SP}}$ pin to ground for the slave, or by an indication in the ICW 4 word, if the buffer mode has been selected ($\overline{\text{EN}}$).

In the case of the master

A0	7							0
1	S7	S6	S5	S4	S3	S2	S1	S0

If a slave interrupt controller is connected to an interrupt input, the bit S_i corresponding to the input to which the (one or more) slaves are attached is set to 1. Otherwise, a 0 denotes that the input is a single request input.

In the case of the slave

A0	7							0
1	0	0	0	0	0	ID2	ID1	ID0

ID2, ID1 and ID0 make up the slave identifier number. The slave will be active when it recognises this number on its three cascade lines.

ICW 4

A0	7							0
1	0	0	0	SFNM	BUF	M/$\overline{\text{S}}$	EOI	1

SFNM (Special Fully Nested Mode): This priority mode is selected by a logic 1 (see priorities below).

BUF	M/$\overline{\text{S}}$	
0	X	Unbuffered mode
1	0	Buffered mode: slave circuit
1	1	Buffered mode: master circuit

Unbuffered mode. Pin $\overline{\text{SP}}$ is hardwired to indicate whether the 8259A is a master or a slave.

Buffered mode is necessary when the 8259A is not connected to the local bus and in the absence of the DEN signal must enable the 8286/8287 transceivers in order to allow the interrupt type to be passed to the CPU during the second $\overline{\text{INTA}}$.

A logical NAND function is required to ensure the logical combining of DEN coming from the 8288 and $\overline{\text{EN}}$ from the 8259A for the transceivers.

EOI (End of Interrupt)

At 1: The automatic end of interrupt mode is selected and the appropriate ISR bit is reset to 0 at the end of the second $\overline{\text{INTA}}$.

At 0: Selection of specific mode - a special command word must be sent before the return from the interrupt procedure.

Commands

These commands can be sent in any order and at any time.

OCW 1

Interrupt masks

A0		7							0
1		M7	M6	M5	M4	M3	M2	M1	M0

M_i: If a bit of the mask word is set to 1, the interrupt request appearing on the corresponding IR_i pin is inhibited. In the reverse case, it is enabled.

In all cases, the requests are written into the IRR.

OCW 2

A0		7							0
0		R	SL	EOI	0	0	L2	L1	L0

R	SL	EOI	
0	0	1	End of interrupt - non specific
0	1	1	*End of interrupt - specific
1	0	1	Priority rotation at time of non-specific EOI

R	SL	EOI	
1	0	0	Rotation authorised at each automatic EOI
0	0	0	Halt rotation on automatic EOIs
1	1	1	*Rotation on each specific EOI
0	1	0	*Fixes the highest priority bit
0	1	0	No operation

*Note that the combinations marked with an asterisk are used in conjunction with bits L2, L1 and L0; the others ignore them.

L2, L1 and L0 may be used to fix the interrupt number involved by the action of OCW 2.

OCW 3

A0	7							0
0	0	ESMM	SMM	0	1	P	RR	RIS

ESMM (Enable Special Mask Mode). If at 1, it enables SMM; if not, the latter is ignored.

SMM (Special Mask Mode)
1 = special mode is selected
0 = special mode is halted.

While an interrupt is being serviced it normally inhibits all those of a lower priority from interrupting. This can cause problems when a particular service routine takes a long time to execute. For example, a service routine may perform an important task early on, followed by extensive low priority tasks. The special mask mode provides a mechanism by which it is possible to accept lower priority tasks before the end of the service routine. In order to achieve this, the following sequence of operations must be executed

1. the bit corresponding to the current routine alone must be masked;
2. the 8259A must be placed in special mask mode using OCW 3;
3. interrupts are enabled, allowing interrupts with a lower priority than the current service routine to be recognised.

Before exiting the service routine, the following sequence must be executed

inhibit the special mask mode;
re-establish normal masking.

P (Poll command)
This type of command is selected by setting bit P to 1. In this mode, the CPU has the responsibility for processing interrupts. These are received as normal by the 8259A, but it cannot interrupt the CPU (flag IF = 0).

When required, the CPU issues a command word OCW 3, containing a 1 in bit P.

The 8259A then handles the next read ($\overline{RD}$ = 0, $\overline{CS}$ = 0) as if it were an interrupt acknowledge cycle.

During this read the 8259A responds by putting the following status word on to the bus.

7							0
I	X	X	X	X	W2	W1	W0

W2-W0 is the binary code of the highest priority interrupt request.

I is equal to 1 when an interrupt has been requested.

This mode has the following benefits

- the interrupts are handled at the moment most convenient for the program, only allowing the desired interrupts to pass through. A simple authorisation (IF = 1) would not allow such a control.
- a procedure that is identical for several interrupt levels is only written once.
- the number of levels can be extended beyond 64 (the maximum number available with cascaded 8259A´s), with the CPU polling as many 8259A´s as required.

RR (Read Register) enables the reading of IRR or ISR registers, if it is set to 1.

The choice of register read is carried out by RIS.

RIS (Read In Service)
This bit is used to select ISR or IRR.

RIS = 1 : the service register ISR is selected
RIS = 0 : the request register IRR is chosen.

This selection will remain effective until another command word is sent, allowing repeated register reads.

Mask read

The interrupt mask, loaded on initialisation, can be carried directly without previously sending a command word, as in the case of ISR and IRR. It is sufficient to carry out a read ($\overline{RD} = 0$, $\overline{CS} = 0$) with A0 at level 1.

It is useful to be able to read the current mask whenever a bit of the mask is modified. This allows a particular bit of the mask to be modified with the aid of an OR instruction to set it or unmask it with an AND instruction, without changing the other mask bits. This new byte may then be written into the mask register of the 8259A.

Priority organisation

Without going into the details of these organisations, it is possible to define the different possibilities.

Fully Nested Mode

This is the normal implicit mode. After the initialisation sequence, IR0 has the highest priority and IR7 the lowest.

Special Fully Nested Mode

When the 8259A controllers are cascaded, the master can be programmed with this mode (ICW 4).

The advantage of this latter mode over the normal mode is that a slave which already has an interrupt in service would be able to inform the master of other inputs to the same slave, if they are of a higher priority.

In normal mode, the master ignores subsequent interrupts from the same slave, up to the EOI command corresponding to the interrupt being serviced.

Consequently, the CPU must verify, before exiting from the interrupt program, that the slave which has interrupted it does not have a second interrupt being serviced.

For that, it sends a non-specific EOI command to the slave and reads its ISR register.

If the latter is empty, it also sends an EOI to the master.

If there is still an interrupt being serviced, it must not send EOI commands to the master.

The automatic end of interrupt must not be chosen in this case.

Priority rotation

Examination of bits R, SL and EOI of the command word OCW 2 indicates the possible choices for modifying priorities by rotation, in order to prevent an interrupt from monopolising the CPU´s attention to the detriment of the others.

In a general way, the lowest priority is assigned to the last interrupt taken into account.

This rotation can be carried out when the EOI is sent, whether automatic, specific or non-specific.

It can also be carried out when OCW 2 (R = 1, SL = 1, EOI = 0) is sent.

Example 5.1

```
PAGE    62,132
TITLE   TEST_NMI
;
;       This program shows a possible use of NMI, allowing the  CPU to exit  from
;an endless wait, when the interrupts are masked. This condition is often met:for
;example,  when trying to establish communication with a peripheral that gives no
;answer,  mismanagement of computing errors of the 8087, endless loop, etc.
;
;       A bounceless  interrupt source  must however be  provided, to  explicitly
;force pin 17 (NMI) of the CPU to 5 volts.
;
;       At the  start of the main  program, a call  for the  INIT_NMI  subroutine
;makes provision for the saving of the actual context as well as  the loading  of
;the type 2 pointer (NMI).
;
;       During an unmaskable  interrupt, this context is  restored and the execu_
;tion starts immediately all over again at the return address of INIT_NMI.
;
;       A certain disturbance may however result if files have been opened; their
;closing must then be planned in the answering procedure to the NMI interrupt.
;
;       Calls to  system are made in  accordance with versions  1.25 and 2.11  of
;MS_DOS.

;
                NAME    TEST_NMI
;               ===============

                EXTRN   INIT_NMI:FAR

        MS_DOS_CALL     EQU     21H
        PROGRAM_END     EQU     4CH
        SCREEN_VIEW     EQU     02H
;
  SG_STACK  SEGMENT  STACK
; ========================
                DW      500 DUP(?)              ; Store reserved for general use.
                DW      128 DUP(?)              ; Supplementary  reservation  for
                                                ; system calls.
   STACK_TOP    LABEL   WORD
;
  SG_STACK  ENDS

  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; ========================================
   VAR_TEST     DB      ?
;
  DATA_TABLE  ENDS
```

```
    CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
  ; ==================================
           ASSUME  CS:CODE_PP,DS:DATA_TABLE

  ;.......Area of constants :.............................................

     MESSAGE      DB      0AH,0DH,'LAST TRIAL'
     MESSAGE_END  LABEL   BYTE
  ;

  ;.......Initialisation of the stack and the segments :........................
  ;
     START:                                     ; START defines together with END
                                                ; START  the entry  point of  the
                                                ; program.
                  MOV     AX,SG_STACK           ; Initialisation of the stack.
                  MOV     SS,AX                 ; Protects the next instruction.
                  MOV     SP,OFFSET STACK_TOP   ; No possible interrupt.
                  MOV     AX,DATA_TABLE         ; Tallying with ASSUME.
                  MOV     DS,AX

                  MOV     VAR_TEST,0            ; VAR_TEST distinguishes the first
                                                ; passage and tests the restart.

  ;.......Storage and initialisation of NMI :..................................

                  CALL    INIT_NMI

  ;.......Possible restart at this point :.......................................

                  CMP     VAR_TEST,055H         ; (CoMPare) After a restart, VAR_
                                                ; TEST is not equal to zero.
                  JE      PASS_2                ; (Jump if Equal)

  ;.......Endless loop caused at first passage :...............................

     PASS_1:      MOV     VAR_TEST,55H
                  MOV     AH,SCREEN_VIEW        ; Screen display of a passage in_
                  MOV     DL,'*'                ; dicator ('*').
                  INT     MS_DOS_CALL           ; Internal interrupt of type  21H
                                                ; used by the system.
                                                ; AH contains the call number and
                                                ; DL contains the character to be
                                                ; transmitted.
     INF_LOOP:    CMP     VAR_TEST,55H
                  JE      INF_LOOP              ; The equality is always confirmed

  ;.......Display of message and return :......................................

     PASS_2:      MOV     CX,MESSAGE_END-MESSAGE ; Subtraction of  offsets  giving
                                                ; the length  of the  message  in
                                                ; bytes.
                  MOV     AH,SCREEN_VIEW
                  XOR     SI,SI                 ; Index initialisation.

     SEND:        MOV     DL,MESSAGE[SI]        ; Display of character pointed to
                                                ; by SI
                  INT     MS_DOS_CALL
                  INC     SI
                  LOOP    SEND                  ; Return to SEND : as long as  CX
                                                ; is not equal to zero.
                  MOV     AL,0
                  MOV     AH,PROGRAM_END
                  INT     MS_DOS_CALL           ; Return to system.

 CODE_PP  ENDS
;
                  END     START
```

Example 5.1a

```
PAGE    62,132
TITLE   INIT_NMI
;
;       This module  contains two  procedures. The first  (CALL_NMI) answers  the
;interrupt by restoring the stack, reloading the essential registers and  substi_
;tuting the return address of INIT_NMI procedure to its own return address.
;
;       Flags must be added since it is an interrupt procedure.
;
;       Also, the mask contained in the interrupt controller 8259A is restored.
;
;       The second procedure (INIT_NMI) is the one that executes all the necessa_
;ry storage.
;
;       Putting CALL_NMI first will allow the request for  address made by  INIT_
;NMI for loading the pointer in the interrupt table to be answered. However,  the
;logical process starts with INIT_NMI.


                NAME    INIT_NMI
;               ================

                PUBLIC  INIT_NMI
;
        PARAM           STRUC
;       --------------------
        CALLING_DS      DW      ?
        FORMER_BP       DW      ?               ; <=BP contains this offset which
        CALLING_IP      DW      ?               ; is the  location of the  return
        CALLING_CS      DW      ?               ; address ±2. (See storage of the
                                                ; stack)
;
        PARAM           ENDS
;
        P               EQU     [BP - OFFSET FORMER_BP]
;
        AFTER_NMI       STRUC
;       --------------------
        IP_RESTART      DW      ?               ; <=BP contains this offset.
        CS_RESTART      DW      ?
        FLAGS           DW      ?
;
        AFTER_NMI       ENDS
;
        O               EQU     [BP]
;
  ADD_INT   SEGMENT   AT 0
; ======================
                        ORG     2*4
   NON_MASKABLE_OFF     DW      ?               ; Location of the pointer for the
   NON_MASKABLE_SEG     DW      ?               ; type 2 (NMI) interrupt.
;
  ADD_INT   ENDS

  PERIPH   SEGMENT   AT 0E000H
; ==========================
   PIC_A0_LOW           DB      1 DUP(?)
   PIC_A0_HIGH          LABEL   BYTE
   PIC_MASK             DB      1 DUP(?)
;
  PERIPH   ENDS
;
```

```
  DATA_TABLE  SEGMENT  WORD  PUBLIC  'RAM'
; =====================================
    SS_BUFFER        DW    ?             ; Storage area of  registers. The
    SP_BUFFER        DW    ?             ; other  registers  may  also  be
    DS_BUFFER        DW    ?             ; stored,depending on the context
    ES_BUFFER        DW    ?
    IP_BUFFER        DW    ?
    CS_BUFFER        DW    ?
    BP_BUFFER        DW    ?
    FLAGS_BFR        DW    ?
    MASK_BFR         DB    ?
;
  DATA_TABLE  ENDS

  CODE_INT  SEGMENT  BYTE  PUBLIC  'ROM'
; ===================================
           ASSUME  CS:CODE_INT,DS:DATA_TABLE
;
  CALL_NMI      PROC    FAR
; ----------------------

                MOV     AX,DATA_TABLE        ; In conformity with ASSUME.
                MOV     DS,AX

;.......Restoring of mask to PIC :..............................................

                MOV     AX,PERIPH            ; No possible  direct  memory  to
                MOV     ES,AX                ; memory transfer.
                MOV     AL,MASK_BFR          ; Transfer of 8 bits with the me_
                MOV     ES:PIC_MASK,AL       ; mory mapped  peripheral  at  ad_
                                             ; dress E0001.
;.......Reordering of the stack :...............................................

                MOV     SS,SS_BUFFER         ; Shields the following  instruc_
                                             ; tion.
                MOV     SP,SP_BUFFER         ; This   instruction   cannot  be
                                             ; interrupted.
                MOV     BP,SP                ; BP allows indirect addressing.

;........Reordering of the return address and flags :...........................

                MOV     AX,IP_BUFFER
                MOV     Q.IP_RESTART,AX
                MOV     AX,CS_BUFFER
                MOV     Q.CS_RESTART,AX
                MOV     AX,FLAGS_BFR
                MOV     Q.FLAGS,AX

;.......End of restoring registers and return of the interrupt :................

                MOV     BP,BP_BUFFER
                MOV     ES,ES_BUFFER
                MOV     DS,DS_BUFFER

                IRET
;
  CALL_NMI      ENDP
;

  INIT_NMI      PROC    FAR
; ----------------------
;.......Entry procedure :.......................................................

                PUSH    BP
                MOV     BP,SP
                SUB     SP,OFFSET FORMER_BP
                MOV     P.CALLING_DS,DS          ; Temporary storage of DS.
                MOV     AX,DATA_TABLE            ; In conformity with ASSUME.
                MOV     DS,AX

;.......Storage of registers :...............................................

                MOV     AX,P.CALLING_DS          ; No possible direct  transfer
                MOV     DS_BUFFER,AX             ; from memory to memory.
                MOV     AX,P.CALLING_IP          ; Storage of the re_start address
                MOV     IP_BUFFER,AX
                MOV     AX,P.CALLING_CS
                MOV     CS_BUFFER,AX
                MOV     AX,P.FORMER_BP
                MOV     BP_BUFFER,AX
                MOV     SS_BUFFER,SS             ; Storage of actual stack.
                MOV     SP_BUFFER,BP             ; BP points the  location of  the
                                                 ; return address by reserving  an
                                                 ; additional word for flags.
                MOV     ES_BUFFER,ES
                PUSHF                            ; Only method of reading the flag
                POP     FLAGS_BFR                ; word.

;.......Storage of the mask of interrupt controller :.................................

                MOV     AX,PERIPH
                MOV     DX,ES                    ; Temporary storage: ES within DX
                MOV     ES,AX
                MOV     AL,ES:PIC_MASK           ; Transfer of 8 bits with the me_
                MOV     MASK_BFR,AL              ; mory mapped  peripheral at  ad_
                                                 ; dress  E0001.
;.......Loading of the NMI pointer :..........................................

                MOV     AX,ADD_INT               ; ES = 0000 is used  to point the
                MOV     ES,AX                    ; interrupt table.
                LEA     AX,CALL_NMI              ; Loading of the offset by  using
                MOV     ES:NON_MASKABLE_OFF,AX   ; ASSUME in order to check access
                MOV     ES:NON_MASKABLE_SEG,CS   ; with CS (group is possible).
                MOV     ES,DX                    ; Restoring of the segment ES.

;.......Return to calling program :...........................................

                MOV     DS,P.CALLING_DS
                MOV     SP,BP
                POP     BP
                RET

;
  INIT_NMI      ENDP
;
   CODE_INT   ENDS
;
                END
```

6 Instruction Set

The instructions can be grouped in six categories:

data transfer
arithmetic operations
logic operations
repetitive instructions (strings)
branch instruction
processor control

The instructions in the first three categories concern the manipulation of the following:

- registers or memory locations to/from registers;
- immediate data to registers or memory locations;
- from the accumulator to/from registers, memory locations or ports.

J.-M. Trio, *8086–8088 Architecture and Programming*
© English Language edition, Macmillan Publishers Ltd 1985

6.1 Data Transfer

Table 6.1 Data Transfer

		Flags									Time (5 MHz)
	General	O	D	I	T	S	Z	A	P	C	µs*
MOV	Transfers a word/byte										0.4-4.4
PUSH	Writes word to top of stack										2-5.6
POP	Reads word at top of stack										1.6-5.8
XCHG	Exchanges bytes or words										0.6-5.8
XLAT or XLATB	Translates byte using a table										2.2
	Input/Output	O	D	I	T	S	Z	A	P	C	
IN	Input byte/word										1.6-2
OUT	Output byte/word										1.6-2
	Address transfer	O	D	I	T	S	Z	A	P	C	
LEA	Load effective address										1.6-2.8
LDA	Load pointer using DS										3.2-5.6
LES	Load pointer using ES										3.2-5.6
	Flag transfer	O	D	I	T	S	Z	A	P	C	
LAHF	Transfer 5 low flags to AH										0.8
SAHF	Transfer AH to 5 low flags					R	R	R	R	R	0.8
PUSHF	Save flags to stack										2
POPF	Restore flags from stack	R	R	R	R	R	R	R	R	R	2

R Restored according to previous value

* Time for the 8086; add 0.8 µs to each word transfer for the 8088

These instructions can be divided into four classes, discussed below.

6.1.1 Explicit transfer

Almost all of these instructions use two operands, and are the only instructions (except for XCHG) that allow a segment register to be used as an operand.

MOV shifts either a byte or a word from a source to a destination. The source and destination may be either a register or memory, with the destination specified before the source with the normal mnemonic.

PUSH saves the word specified by the source operand on the top of the stack. The stack pointer is decremented by 2 before the transfer. The source can be a memory location (see examples 4.13 and 4.15a).

POP restores the word located at the top of the stack, addressed by SP, to the destination operand. The stack pointer is incremented by 2 after the transfer. The destination can be a memory address (see examples 4.13 and 4.15a).

XCHG exchanges the byte or word of the source operand with that of the destination operand. Segment registers cannot be XCHG operands. However, a word or a byte located in memory can be exchanged with a register.

6.1.2 Implicit transfer with the accumulator

IN transfers a byte (or a word) from the input port to the AL (or AX) register. The port is specified either with aid of a byte placed in the instruction as an immediate value and allowing access to ports 0 to 255, or with the aid of the number of the port contained in the DX register and allowing access to any of the 64K port addresses.

OUT is the reverse of IN and transfers data from the accumulator to the output port specified in the same way as IN.

XLAT or **XLATB** allows the translation of a byte with the aid of a table composed of bytes. Register AL serves as an index into the table, consisting of a maximum of 256 bytes, whose base is addressed by BX. The byte operand, thus selected, is transferred to AL. The name of the table, which is specified in the instruction, serves only to select the segment register as a function of the current ASSUME and to verify the type. If this name is omitted (XLATB), DS is used implicitly and the type is not verified (example 4.19a).

6.1.3 Address transfers

LEA (Load Effective Address) transfers the offset of the source operand to the destination operand. The source operand must be a memory operand. The destination operand must be a 16-bit register (general, pointer or index). The offset is calculated automatically from the base of a group if ASSUME is effective (see examples 4.8, 4.12 and 6.1).

LDS transfers the source address (contained in 32 bits) to two destination registers. The address of the segment is transferred into DS while the address of the offset is transferred into a 16-bit register - general, pointer or index (see examples 3.4, 4.15a).

LES has a role similar to that of LDS but the address of the segment is transferred into ES (instead of DS) (see example 4.4).

6.1.4 Status word transfer

LAHF transfers the low byte of the status word, containing SF, ZF, AF, PF and CF into register AH, without modifying their position (bits 7,6,4,2 and 0 respectively).

AH :	S	Z	-	A	-	P	-	C

SAHF carries out the opposite operation. It transfers the contents of AH into the low byte of the status word (SF, ZF, AF, PF AND CF). The format of AH must be the same as for LAHF.

PUSHF decrements the SP register by 2 and transfers the complete status word into the stack location addressed by SP, taking account of the flag positions in the word:

-	-	-	-	O	D	I	T		S	Z	-	A	-	P	-	C

(See example 5.1a.)

POPF transfers the specific bits of the stack element addressed by SP into the flag registers and then decrements SP by 2. The format is the same as for PUSHF.

6.2 Arithmetic Operations

The 8086/8088 instruction set makes provision for computation of the four basic mathematical functions. It is possible to carry out 8 or 16 bit operations on signed and unsigned numbers.

These operations set flags that may be used to control conditional jumps. Provision is also made to modify the results of these operators to provide results in decimal (BCD) format and even to form extended decimal numbers in which each BCD digit is preceded by a 0 in order to facilitate easy conversion to the corresponding ASCII code.

6.2.1 Addition

ADD adds the source operand and the destination operand and writes the result in the destination operand.

ADC adds source and destination operands, adds 1 if the flag (CF) is set to 1 and returns the result into the destination operand. This instruction allows double length (or longer) calculations using carry.

INC increments the operand by 1 without setting the carry flag (CF). It may modify the overflow flag (OF) (see example 4.9).

AAA allows additions involving two decimal numbers of extended format to be corrected such as to obtain a result of the same type. This instruction updates the AF and CF flags, thus allowing double length calculation in this format. The instruction operates implicitly on AL.

DAA allows additions involving two decimal numbers to be corrected such as to obtain a result in the same format. The flags are normally set, with the exception of OF which is undefined. The instruction operates implicitly on AL.

6.2.2 Subtraction

SUB subtracts the source operand from the destination operand and returns the result to the destination operand (see examples 4.13 and 4.15a).

SBB subtracts the source from the destination, takes 1 from the result if the CF flag is set to 1 and writes the result in the source operand.

DEC decrements the operand by 1, but does not set the carry flag (see example 4.8).

Table 6.2 Arithmetic Instructions

		Flags									Time (5 MHz) μs(a)
	Addition	O	D	I	T	S	Z	A	P	C	
ADD	Addition of bytes/words	*				*	*	*	*	*	0.6-5.8
ADC	Addition of bytes/words with carry	*				*	*	*	*	*	0.6-5.8
INC	Increment a byte/word by 1	*				*	*	*	*		0.4-5.4
AAA	ASCII adjust for addition	X				X	X	*	X	*	0.8
DAA	Decimal adjust for addition	*				*	*	*	*	*	0.8
	Subtraction	O	D	I	T	S	Z	A	P	C	
SUB	Subtraction of bytes or words	*				*	*	*	*	*	0.6-5.8
SBB	Subtraction of bytes or words with borrow	*				*	*	*	*	*	0.6-5.8
DEC	Decrement byte/word by 1	*				*	*	*	*		0.4-5.4
NEG	Two´s complement byte/ word (change sign)	*				*	*	*	*	1b	0.6-5.6
CMP	Compare bytes or words	*				*	*	*	*	*	0.6-4.4
AAS	ASCII adjust for subtraction	X				X	X	*	X	*	0.8
DAS	Decimal adjust for subtraction	X				*	*	*	*	*	1
	Multiplication	O	D	I	T	S	Z	A	P	C	
MUL	Multiply bytes or words (unsigned)	*				X	X	X	X	*	14-30.2
IMUL	Integer multiply bytes or words (signed)	*				X	X	X	X	*	16-34.4
AAM	ASCII adjust for multiplication	X				*	*	X	*	X	16.6
	Division	O	D	I	T	S	Z	A	P	C	
DIV	Divide bytes or words (unsigned)	X				X	X	X	X	X	16-36
IDIV	Integer divide bytes or words (signed)	X				X	X	X	X	X	20.2-40.4
AAD	ASCII adjust for division	X				*	*	X	*	X	12
CBW	Convert byte to word										0.4
CWD	Convert word to double word										1

* Flag modified according to result

X Undefined

(a) Add 0.8 μs for each word transfer for the 8088

(b) 0 if the number is zero

NEG negates the number by carrying out the two´s complement plus 1.

CMP subtracts the source from the destination in order to set the flags but leaves both source and destination unchanged. This instruction is generally effected before making conditional jumps (see examples 4.8, 4.19a and 6.1).

AAS allows subtractions involving two bytes containing decimal numbers in extended format to be corrected so as to obtain a result of the same type. Chained calculations are possible in this format, and the instruction operates implicitly on AL.

DAS allows subtractions involving two bytes containing decimal numbers to be corrected so as to obtain a result of the same type, in AL. Chained calculations are possible.

6.2.3 Multiplication

MUL carries out an unsigned multiplication between the contents of the accumulator (AL or AX) and the source operand. The double length result is written to the accumulator and its extensions: AX and DX for a 8-bit and 16-bit operation respectively. CF and OF are set to 1 if the AH (8 bit) or DX (16 bit) register is not zero, so as to indicate that they contain a significant result (see examples 4.8 and A.1a).

IMUL carries out a signed integer multiplication in a manner similar to MUL.

AAM allows multiplications involving two bytes and containing expanded format decimal numbers to be corrected so as to obtain a result of the same type. The operation necessarily involves AL and AH.

6.2.4 Division

DIV carries out an unsigned division on the accumulator and its extension (AL and AH for a 8-bit operation; AX and DX for a 16-bit operation) by the source operand. It writes the quotient in the accumulator (AL or AX) and the remainder in the accumulator extension (AH or DX). The status of the flags is undefined after this operation. Division by 0 causes a type 0 interrupt (which is non-maskable like all internal interrupts).

IDIV carries out a signed integer division in a manner similar to DIV.

AAD allows divisions involving two bytes containing extended format decimal numbers to be corrected so as to obtain a result of the same type. This operations is carried out, on AL, before division, in contrast to the other corrections. AH must be at zero before the operation.

CBW converts the signed byte contained in AL into a word of the same sign in AX. This instruction is often assigned to a division.

CWD converts the signed word contained in AX into a double word of the same sign in AX and DX. This instruction is often assigned to a division.

6.3 Logic Operations

The 8086 carries out basic logic operations on 8 and 16 bit operands.

NOT carries out the one´s complement without setting the flags.

SHL, SAL, SHR, SAR. These shift operations are carried out on memory words or registers, and allow the contents to be shifted to the left or to the right. Zeros are shifted into the end of the word, except in the case of signed SAR. The number of places to be shifted is held in CL, if it is greater than 1, or, if it is equal to 1, may be written directly in the instruction. CF contains the last bit to be shifted out from the operand. OF, which is defined only for 1-bit shifts, is set to 1 if the final value of the sign bit differs from the initial value. PF, SF and ZF are set to 1 depending on the resulting value (see examples 3.9 and 4.19a).

ROL, ROR, RCL, RCR. The ROL or ROR operations are carried out on memory words or registers, and allow rotation to the left or to the right. Rotation occurs when the bit at one end is transferred to the other end. The shift number is either 1 or contained in CL.

For RCL and RCR, rotation is carried out in the same way, but passing through CF first.

The flags are set in the same way as for shift operations.

AND carries out the logical AND between the source and destination operands. The result is written in the destination (see example 4.9).

TEST has a similar role to AND but does not write the result. Only the flags are set. This instruction is very useful to test the state of particular bits of a byte or a word.

OR carries out the logical OR between the source and destination operands. The result is written in the destination operand.

XOR similarly carries out the logical exclusive OR (see examples 4.8 and 5.1).

Table 6.3 Logic Operations

		Flags									Time (5 MHz)
	Logic	O	D	I	T	S	Z	A	P	C	µs(a)
NOT	One´s complement of a byte or word										0.6-5.6
AND	Logical AND of two bytes or two words	0				*	*	X	*	0	0.6-5.8
OR	Logical OR of two bytes or two words	0				*	*	X	*	0	0.6-5.8
XOR	Exclusive logical OR of 2 bytes/2 words	0				*	*	X	*	0	0.6-5.8
TEST	AND comparison of bytes or words	0				*	*	X	*	0	0.6-4.6
	Shift	O	D	I	T	S	Z	A	P	C	
SHL/ SAL	Shift logical or arithmetic left (b/w)	*								*	0.4-6.4b
SHR	Shift logical right (byte or word)	*								*	0.4-6.4b
SAR	Shift arithmetic right (byte or word)	*				*	*	X	*	*	0.4-6.4
	Rotation	O	D	I	T	S	Z	A	P	C	
ROL	Rotate left (b or w)	*								*	0.4-6.4b
ROR	Rotate right (b or w)	*								*	0.4-6.4b
RCL	Rotate through carry flag left (b or w)	*								*	0.4-6.4b
RCR	Rotate through carry flag right (b or w)	*								*	0.4-6.4b

*	Flag altered according to the result
X	Undefined
(a)	Add 0.8 µs for each word transfer for the 8088
(b)	Add 0.8 µs per shift, starting with the second

6.4 Repetitive Instructions (Strings)

Simple instructions allow various manipulations on strings of bytes or words.

Each of these instructions can be executed in a repetitive manner by preceding the instruction with a repeat prefix (for example, REP). Instructions may also be combined together to perform complex string manipulations.

6.4.1 Operating mode

All instructions that concern strings use the SI register to address the source operand and the DI register to point to the destination. The names of tables that are specified in the instruction only serve for the choice of type and segment (see below).

DI is necessarily assigned to the segment register ES. On the other hand, the segment assigned to SI, implicitly DS, can be modified as a function of the current ASSUME.

If the direction flag DF is set to 0, the pointers are incremented after each instruction, by 1 for a movement of bytes, by 2 for words.

If the DF flag is set to 1, the pointers are decremented.

All these instructions can be preceded by a REP prefix (occupying 1 byte), which indicates that the operation must be repeated for as long as CX is greater than 0.

CX specifies the required number of repeats and is decremented by 1 (whatever the type) after each repeat, until a zero value is reached.

Thus, an initial value of 0 in CX would not initiate any execution.

The prefix byte REPZ or REPNZ indicates that a test on ZF must be carried out at each iteration.

If the instructions sets the ZF flag and if ZF is different from the state indicated by the prefix, the repetition is halted.

The instruction JCXZ (jump if CX is zero) placed at the exit of a loop allows checking for an exit made before the end of the count.

Repetitive instructions can be interrupted at the start each new sequence. When the program returns to the instruction, the effect of the REP prefix, placed immediately in front of it, is taken into account and the string transfer can continue normally.

Note that the same does not apply if other prefixes are also present, such as LOCK or a segment corrector prefix. These will not be taken into account on return from an interrupt.

In such cases, interrupts have to be inhibited, even at the risk of preventing the CPU from fulfilling its own important requirements during the complete transfer time or accepting a non-maskable interrupt.

It is therefore preferable to make systematic use of the DS segment register to point to the source segment and thus to avoid use of the segment corrector prefix.

In this case, the shortened repetitive instructions, for example MOVSB or MOVSW, can be used, with the byte (B) or word (W) type being indicated by the additional letter (see examples 6.1a and A.1a).

6.4.2 Base instructions

Table 6.4 String Instructions

		Flags									Time (5 MHz) µS(a)
	Prefix	O	D	I	T	S	Z	A	P	C	
REP	Repeat while CX not 0										0.4
REPE/ REPZ	Repeat while equal and CX ≠ 0										0.4
REPNE/ REPNZ	Repeat while not equal and CX ≠ 0										0.4
	Instruction	O	D	I	T	S	Z	A	P	C	
MOVS/ MOVSB/ MOVSW	Move byte or word string										4-5.6b
CMPS/ CMPSB/ CMPSW	Compare byte or word string	*				*	*	*	*	*	4.8-6.6c
SCAS/ SCASB/ SCASW	Scan byte or word string	*				*	*	*	*	*	3.2-5d
LODS/ LODSB/ LODSW	Load byte or word string into AL or AX										2.6-4.6e
STOS/ STOSB/ STOSW	Store byte or word string from AL or AX										2.4-4f

* Flag altered according to the result
(a) Add 0.8 µs for each word transfer for the 8088
(b) Add 3.8 µs for each iteration
(c) Add 4.8 µs for each iteration
(d) Add 3.2 µs for each iteration
(e) Add 2.8 µs for each iteration
(f) Add 2.2 µs for each iteration

MOVS transfers a byte (or word) operand, pointed to by SI, from the source (on the right) to the destination (on the left), pointed to by ES:[DI]. Using the REP prefix, this is equivalent to moving a memory block to another location (see example A.1).

CMPS subtracts the byte or word destination from the source and sets the flags without altering the operands. ES:[DI] points to the destination and SI points to the source.

Using the REPE prefix, it is possible to determine whether two blocks are identical and therefore perhaps find the first different element. In contrast, REPNE is used to find the first identical element from two blocks.

SCAS subtracts the byte or word addressed by ES:[DI] from the contents of AL or AX respectively; it also sets the flags without altering the operands. When used in conjunction with REPE, it allows a different element to be found in a block whose elements are usually all equal to the contents of AL or AX.

Used with REPNE, the first occurrence of the value contained in AL or AX found in a block is identified.

LODS transfers the byte or word addressed by SI into AL or AX. This operation normally does not use the REP prefix, but can be placed in a loop.

STOS transfers the contents of AL or AX into the byte or word addressed by ES:[DI]. When used with the REP prefix, this instruction allows a block to be filled with the same value for all the elements (see example A.1a).

For these operations, the writing of operands is only done to allow the assembler to verify the accessibility of variables, as well as their type.

A letter B (byte) or W (word) allows operands to be omitted (see earlier note).

In all cases, the source operand is addressed by SI and the destination by DI.

It is only in the instructions CMPS and MOVS that the operands of the two types are used at the same time.

6.5 Jump Instructions

There are four possible types of program transfer

unconditional tranfers
conditional transfers
iteration control
interrupts

All these transfers cause program fetches to resume from a new memory location, sometimes even in a new segment. Conditional transfers are made within a range of -128 to +127 bytes from the transfer instruction.

Table 6.5 Transfer Instructions

		Flags									Time (5 MHz)
	Unconditional transfers	O	D	I	T	S	Z	A	P	C	μs[a]
CALL	Call procedure										3.4-10.6
RET	Return from procedure										1.8-4
JMP	Jump										2.2-7.6
LOOP	Loop while CX ≠ 0										1/3.4
LOOPE/ LOOPZ	Loop while CX ≠ 0 and ZF = 1										1.2/3.6
LOOPNE LOOPNZ	Loop while CX ≠ 0 and ZF = 0										1/3.8
JCXZ	Jump if CX = 0										1.2/3.6
	Interrupts	O	D	I	T	S	Z	A	P	C	
INT	Internal interrupt			0	0						11.4
INTO	Interrupt if OF = 1 (overflow)			0	0						0.8/11.6
IRET	Interrupt return	R	R	R	R	R	R	R	R	R	5.4

(a) Add 0.8 μs for each word transfer for the 8088
R Restored according to a previous value

6.5.1 Unconditional transfers

These can cause branching within the current segment or into another segment which then becomes the new current segment. Direct and indirect transfers are possible.

CALL saves the offset of the next instruction on the stack. In the case of an intersegment jump the contents of the CS register are saved first. Control is subsequently passed to the address specified as the call operand (see examples 2.16, 4.2 and 4.19a).

JMP transfers control to the called operand without saving IP and CS (see examples 2.6 and 4.2).

RET n transfers control to the return address stored on the stack at the time of the preceding CALL. It can also adjust the SP register so as to recover the locations used for the passing of arguments; ´n´ then indicates the number of bytes occupied by the arguments (see examples 2.14, 2.15 and 4.19a).

6.5.2 Iteration control

These instructions are especially useful when combined with those for string manipulation.

The destination must lie between +127 or -128 bytes from the start instruction.

LOOP decrements the CX count register by 1 and causes a jump if CX is not equal to 0. The next sequential instruction is executed if CX = 0 (see examples 2.8, 3.9 and 4.8).

LOOPZ (or LOOPE) decrements CX by 1. It causes a program transfer if CX is not equal to 0 and if the zero flag (ZF) is set. Otherwise the next instruction is executed (see example 4.2).

LOOPNZ (or LOOPNE) decrements CX by 1. It causes a program transfer if CX is not equal to 1 and if the ZF flag is at 0.

JCXZ causes program transfer if CX is equal to 0. When used at the exit from a loop, this instruction allows the cause of the loop exit to be identified; that is, end of count or action by ZF (see examples 4.2 and 6.1).

6.5.3 Internal interrupts

The control of program execution is transferred in a manner similar to external interrupts coming from an 8259A interrupt controller.

All these interrupts cause a transfer while saving the flag registers on the stack, as for a PUSHF, and carrying out an indirect intersegment CALL to an address obtained from the interrupt table.

The interrupt type is shown in the operand.

This instruction allows execution of a routine normally used by an external interrupt.

INT saves the flags, resets TF and IF, and jumps to the address specified in the appropriate locations of the 256 entry interrupt table. If a type 3 interrupt is involved, a short (1 byte) form of the instruction is generated (see examples 4.19a and 5.1).

INTO works like INT but only if the OF flag is at 1 (overflow).

IRET transfers control to the return address, saved at the time of the preceding interrupt and restores flag registers (like POPF) (see examples 2.15 and 5.1a).

6.5.4 Conditional jumps

These instructions allow a jump depending on the boolean functions of flag registers.

The destination must be between +127 and -128 bytes from the instruction.

Table 6.6 Conditional Transfer Instructions

	Conditional transfers	Flags O D I T S Z A P	Time (5 MHz) µs
JA or	Jump if above		0.8/3.2
JNBE	(if $CF + ZF = 0$)[a]		
JAE or	Jump if above or equal		0.8/3.2
JNB	(if $CF = 0$)[a]		
JB or	Jump if below		0.8/3.2
JNAE	(if $CF = 1$)[a]		
JBE or	Jump if below or equal		0.8/3.2
JNA	(if $CF + ZF = 1$)[a]		
JC	Jump if carry		0.8/3.2
	(if $CF = 1$)		
JE or	Jump if equal or zero		0.8/3.2
JZ	(if $ZF = 1$)		
JG or	Jump if greater		0.8/3.2
JNLE	(if $(SF \oplus OF) + ZF = 1$)[b]		
JGE or	Jump if greater or equal		0.8/3.2
JNL	(if $SF \oplus OF = 0$)[b]		
JL or	Jump if less		0.8/3.2
JNGE	(if $SF \oplus OF = 1$)[b]		
JLE or	Jump if less or equal		0.8/3.2
JNG	(if $(SF \oplus OF) + ZF = 1$)[b]		
JNC	Jump if no carry		0.8/3.2
JNE or	Jump if not equal or not zero		0.8/3.2
JNZ	(if $ZF = 0$)		
JNO	Jump if not overflow		0.8/3.2
	(if $OF = 0$)		
JNP or	Jump if not parity		0.8/3.2
JPO	(if $PF = 0$)		
JNS	Jump if not sign		0.8/3.2
	(if $SF = 0$)		
JO	Jump if overflow		0.8/3.2
	(if $OF = 1$)		
JP or	Jump if parity equal		0.8/3.2
JPE	(if $PF = 1$)		
JS	Jump if sign negative		0.8/3.2
	(if $SF = 1$)		

(a) Relates to unsigned numbers

(b) Relates to signed numbers

Table 6.6 lists the possible instructions, the conditions assigned to them and their interpretation.

Note the use of greater and less than for the signed number, while above and below refer to unsigned numbers (see examples 4.8, 5.1 and 6.1).

6.6 Processor Control

Table 6.7 CPU Control Instructions

		Flags									Time (5MHz)
	Flag operations	O	D	I	T	S	Z	A	P	C	µs
STC	Set carry flag to 1									1	0.4
CLC	Clear carry flag									0	0.4
CMC	Complement carry flag									*	0.4
STD	Set direction flag		1								0.4
CLD	Clear direction flag		0								0.4
STI	Set interrupt enable flag			1							0.4
CLI	Clear interrupt enable flag			0							0.4
	External synchronisation	O	D	I	T	S	Z	A	P	C	
HLT	Halt until interrupt or reset										0.4
WAIT	Wait for TEST pin active										0.6a
ESC	Escape to external processor										0.4-4
LOCK	Lock bus for next instruction										0.4
	No operation	O	D	I	T	S	Z	A	P	C	
NOP	No operation										0.6

(a) Pin TEST tested every 1 µs.

6.6.1 Operations on flags

CLC sets CF flag to 0.

CMC carries out complement of CF flag.

STC sets CF flag to 1.

CLD sets DF flag to 0, thus requiring string instructions to auto-increment the operand pointers (see example 6.1).

STD sets DF flag to 1, thus requiring string instructions to auto-decrement the operand pointers.

CLI sets the IF flag to 0, thus preventing external interrupts (except non-maskable interrupts).

STI sets the IF flag to 1, thus allowing external interrupts after execution of the next instruction.

6.6.2 Processor halt

HLT causes the 8086 processor to halt. This is terminated by an appropriate external interrupt or by a RESET. In practice an infinite loop occurs on the HLT instruction itself. After the interrupt, control is returned to the next instruction since it is its address that was saved on the stack at the time of the HLT instruction.

Note that this instruction may be unusable when an assembler program is executed under the control of the operating system. The latter frequently generates numerous interrupts for its own use (for example, date-time) and would rapidly initiate a jump to the next instruction. An explicit infinite loop including at least one NOP instruction should be used instead.

6.6.3 Processor wait

WAIT causes the processor to wait until its $\overline{\text{TEST}}$ input is at 0. This wait can be interrupted by an appropriate interrupt.

However, in this case the instruction address saved at the time of the jump is the WAIT address. On return, the wait will therefore resume.

6.6.4 Escape

ESC indicates to the CPU that an instruction intended for the numeric co-processor extension (NPX) has been fetched, and requires the CPU to generate the appropriate addresses.

This instruction is normally generated automatically by the assembler on reading a letter F at the start of an instruction for the NPX.

6.6.5 Bus inhibit

A special one-byte prefix that precedes an instruction prevents other bus masters from using the bus during execution of the instruction. This facility is useful in multiprocessor applications (see LOCK) in order to avoid memory overwrites.

6.6.6 Single step

When the TF flag is set to 1, the processor generates a type 1 interrupt after execution of each instruction.

This method of operation is generally used by an emulator system designed to debug the program.

Example 6.1

```
PAGE    62,132
TITLE   PASSWORD
;
;       This program is protected by a  password of six  alphanumeric characters.
;Conceived as a main  program, it must  first of all set up the  stack. Then,  an
;MSDOS function call allows keyboard requests without echoes on the screen. After
;a predefined number of unsuccessfull trials (4 in this case), the process is im_
;mediately aborted. Otherwise, the message 'PASSWORD ACCEPTED' is displayed,  and
;the program is executed.
;
;       The comparison of passwords is achieved using the REPE CMPSB  instruction
;which performs the comparison, increments the index and counts  the  characters.
;On exit the JCXZ instruction is used to identify premature exit from the loop as
;a result of a false character.
;
;       The  system  calls  are  executed  according to  versions 1.25 or 2.11 of
;MS_DOS.

;
                NAME    PASSWORD
;               ================


        MS_DOS_CALL     EQU     21H
        PROGRAM_END     EQU     4CH
        DISPLAY_VIDEO   EQU     02H
        KB_WITHOUT_ECHO EQU     08H
        TRIAL_NUMBER    EQU     4
;
        PARAM           STRUC
;       --------------------
                        DB      ?                ; Allows stack  alignement to  be
                                                 ; kept (only 8086).
        COUNTER         DB      ?                ; Byte counting number of trials.
        TRIAL_PASSWORD  DB      6 DUP(?)         ; Stacking of password on  trial.
        FORMER_BP       DW      ?                ; <= BP points this offset.
;
        PARAM           ENDS
;
        P               EQU     [BP - OFFSET FORMER_BP]
        OFF_LOCATION    EQU     FORMER_BP - TRIAL_PASSWORD
;
  SG_STACK  SEGMENT  STACK
; =======================
                        DW      500 DUP(?)       ; Reservation for general use.
                        DW      128 DUP(?)       ; Supplementary    reservation
                                                 ; for systems calls.
        STACK_TOP       LABEL   WORD
;
  SG_STACK  ENDS
;
```

```
  DIRECTORY   SEGMENT   WORD   PUBLIC   'RAM'
; ====================================
        RIGHT_PASSWORD            DB      'XYZ873'
        PASSWORD_END      LABEL   BYTE
        MESSAGE                   DB      0AH,0DH,'PASSWORD ACCEPTED'
        MESSAGE_END       LABEL   BYTE
;
  DIRECTORY   ENDS

  CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; ====================================
        ASSUME  CS:CODE_PP,ES:DIRECTORY

;.......Initialisation of stack and segments :.................................
;
  START:                                        ; START: fixes the program  entry
                                                ; point with END START.
                MOV     AX,SG_STACK             ; Definition of the stack.
                MOV     SS,AX                   ; Shields the following  instruc_
                                                ; tion.
                MOV     SP,OFFSET STACK_TOP     ; No interrupt possible.
                MOV     AX,DIRECTORY            ; In conformity with ASSUME.
                MOV     ES,AX

                PUSH    BP                      ; BP=> FORMER_BP of the structure
                MOV     BP,SP                   ; BP points to the stack
                SUB     SP,OFFSET FORMER_BP     ; Location of  local variables in
                                                ; the stack.

;.......Request for password :...................................................

                MOV     P.COUNTER,0             ; Counter initialisation.

  TRIAL:        CMP     P.COUNTER,TRIAL_NUMBER  ; Test of number of trials.
                JNE     OTHER_TRIAL             ; Conditional   short  jump  (127
                JMP     ABORT                   ; bytes max) which branches to  a
                                                ; JMP in order to reach ABORT.

  OTHER_TRIAL:  MOV     AH,DISPLAY_VIDEO        ; Screen display of a message re_
                MOV     DL,'?'                  ; quest ('?').
                INT     MS_DOS_CALL             ; Type  21H  internal   interrupt
                                                ; executed by the system. AH con_
                                                ; tains  the call  number and  DL
                                                ; the character to be transmitted

;.......Reading, without echo, of the characters of password on trial :..........

                MOV     CX,LENGTH TRIAL_PASSWORD; LENGTH gives the number of DUP.
                MOV     AH,KB_WITHOUT_ECHO
                XOR     DI,DI                   ; Index initialisation.
  RECEPTION:    INT     MS_DOS_CALL             ; Read character returned to AL.
                MOV     P.TRIAL_PASSWORD[DI],AL ; AL within SS:[BP]+offset TRIAL_
                                                ; PASSWORD+DI
                INC     DI
                LOOP    RECEPTION               ; request as long as CX ≠ 0.
;

;.......Using CMPS for comparing TRIAL_PASSWORD with RIGHT_PASSWORD :............

                LEA     DI,RIGHT_PASSWORD       ; ES:[DI] points the  right  ope_
                                                ; rand of CMPS.ES is compulsory.
                MOV     SI,BP                   ; SI points to the left  operand.
                SUB     SI,OFF_LOCATION         ; Its segment depends on ASSUME.
                MOV     AX,SS                   ; No  named  variable:  anonymous
                MOV     DS,AX                   ; CMPS  must be  used. DS is then
                                                ; compulsory.
                MOV     CX,LENGTH TRIAL_PASSWORD; LENGTH gives the number of DUP.
                CLD                             ; (CLear Direction flag)=> DI and
                                                ; SI  are  increased by 1 (CMPSB)
        REPE    CMPSB                           ; Repetition as  long as CX≠0 and
                                                ; bytes of  strings are the same.
                                                ; This instruction may be  inter
                                                ; rupted  at   every   repetition
                                                ; (REPE being the only prefixe).
                JCXZ    CORRECT                 ; Jump if CX =0

                INC     P.COUNTER               ; Increment counter.
                JMP     TRIAL                   ; Trial again.

;.......Sending of message 'PASSWORD ACCEPTED' :...............................

    CORRECT:    MOV     CX,MESSAGE_END-MESSAGE  ; Offset  substraction   giving
                                                ; length of message in  bytes.
                MOV     AH,DISPLAY_VIDEO
                XOR     SI,SI                   ; Index initialisation.

    SEND:       MOV     DL,MESSAGE[SI]          ; Output  of character indexed by
                                                ; SI.
                INT     MS_DOS_CALL
                INC     SI
                LOOP    SEND                    ; Return  to  SEND : as  long  as
                                                ; CX ≠ 0.

;.......Continuation of program protected by password,.then end :...............

;               .
;               .
;               .
    ABORT:      MOV     AL,0
                MOV     AH,PROGRAM_END
                INT     MS_DOS_CALL             ; Return to system.

CODE_PP   ENDS
;
                END     START
```

7 8087 Numeric Processor Extension

7.1 Functional Description

The 8087 Numeric Processor Extension (NPX), also called Numeric Data Processor (NDP), is a co-processor designed to extend the capabilities of the 8086/8088 in maximum mode. The NPX performs arithmetic and logical operations as well as transcendental functions on numerical variables, and provides full floating point arithmetic support.

The 8087 extends the number of registers and the CPU instruction set and adds new types of variables.

The combination of the 8086/8088 and the 8087 can be viewed as a single processor unit by the programmer.

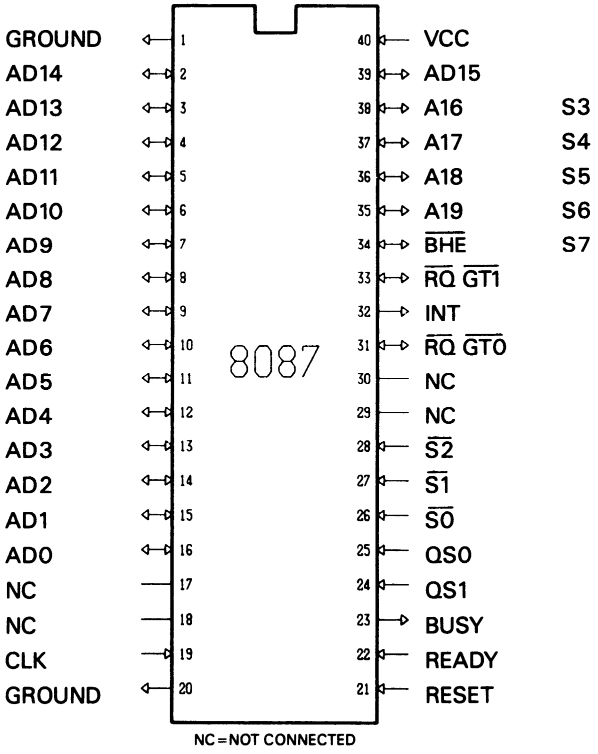

GROUND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NC 17
NC 18
CLK 19
GROUND 20
40 VCC
39 AD15
38 A16 S3
37 A17 S4
36 A18 S5
35 A19 S6
34 $\overline{\text{BHE}}$ S7
33 $\overline{\text{RQ}}$ $\overline{\text{GT1}}$
32 INT
31 $\overline{\text{RQ}}$ $\overline{\text{GT0}}$
30 NC
29 NC
28 $\overline{\text{S2}}$
27 $\overline{\text{S1}}$
26 $\overline{\text{S0}}$
25 QS0
24 QS1
23 BUSY
22 READY
21 RESET
8087
NC=NOT CONNECTED

Figure 7.1

J.-M. Trio, *8086–8088 Architecture and Programming*
© English Language edition, Macmillan Publishers Ltd 1985

7.1.1 System configuration

The 8087 is connected in parallel with the CPU - either 8086 or 8088 - and forms an extension to it.

Its pin arrangement is therefore very close to that of the processor in maximum mode.

The status lines $\overline{\text{S0}}$-$\overline{\text{S2}}$ and QS0-QS1 allow the NPX to follow and decode the instructions being executed by the CPU without affecting the CPU operation.

The 8087 monitors the instructions being fetched by the 8086/8088 and maintains an identical instruction queue. All instructions intended for the NPX have the 11011 binary value in the most significant 5 bits of the first byte of the instruction. When an instruction with this ´ESCAPE´ prefix is detected, it is processed by the NPX instead of the CPU.

In order to synchronise its operation with the 8086/8088, the NPX activates the BUSY signal informing the CPU that it is executing an instruction. The CPU WAIT instruction allows the CPU to test the signal $\overline{\text{TEST}}$ = BUSY to confirm that the NPX has completed its operation.

The NPX can interrupt the CPU when it detects an error or an exception. The NPX interrupt signal is normally routed via an 8259A interrupt controller.

The CPU $\overline{\text{RQ}}$-$\overline{\text{GT1}}$ line is normally assigned to the 8087 to allow it to obtain control of the bus for data transfers.

The $\overline{\text{RQ}}$-$\overline{\text{GT0}}$ line of the CPU remains available for other system requirements (for example, a local IOP).

Another master of the local bus can be connected to the $\overline{\text{RQ}}$-$\overline{\text{GT1}}$ of the NPX itself.

In this case, the 8087 transmits the bus request to the CPU if the CPU is currently master of the bus. If the 8087 has control of the bus, the NPX itself hands over the bus as it always considers that other devices have a higher priority than itself.

For example, two 8089 co-processors can thus be added to the 8086/87 system. One of them shares use of the 8086 bus with the 8087 on the basis of first come first served; the second being assured of priority over the 8087 by connection to the 8087 $\overline{\text{RQ}}$/$\overline{\text{GT1}}$ line.

All local processors must have the same clock and the same bus interfaces.

7.1.2 Sequence of bus operations

The structure, operation and timing of the 8087 buses are identical to those of the 8086/8088 in maximum mode.

The address is multiplexed with the data on the first 16 or 8 lines of the address and data bus.

A16-A19 are multiplexed with status lines S3-S6.

When the 8087 controls the bus, S4 and S6 are always at 1.

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0	X	X	Unused
1	0	0	Unused
1	0	1	Read data in memory
1	1	0	Write data in memory
1	1	1	Passive

When it is monitoring the CPU bus cycles, the state of S6 allows the NPX to differentiate the activities of the 8086/88 (S6 - 0) from those of another bus master.

The 8087 determines the type of the current bus cycle by decoding lines $\overline{S0}$-$\overline{S2}$.

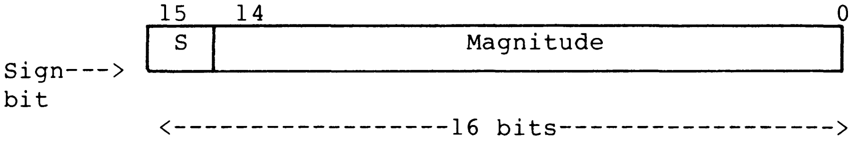
7.1.3 Representation of numbers

Normal numbers

The NPX can read and write seven different forms of numerical representation, grouped as four types in memory.

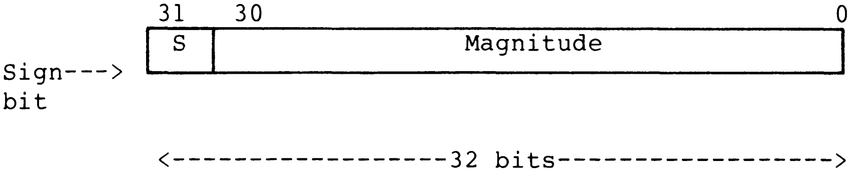
All exchanges of values between the NPX and the CPU are made via the memory.

WORD Integer



The 16-bit number is stored as two´s complement and can take values between -32768 and +32767.

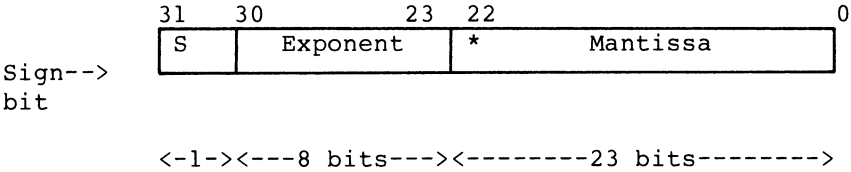
Short integer



The number is stored as 32 bit two´s complement and can take values between the following approximate values

-2×10^9 and $+2 \times 10^9$

Short real format



(* indicates implicit ´1´)

The sign of the real number is written in bit 31, and the exponent is written as an 8-bit binary value with an offset of 127 (= 7FH).

The mantissa is written in the lowest 23 bits, with the implicit ´1´ that should appear in position 24 omitted, because it carries no information. The posssible values range in absolute value from approximately the following

1.2×10^{-38} to 3.4×10^{38}

Example
The value -1 in short real is written

: 1 : 0111 111 : 00----------------------00 :

$= -2^{(127-127)} = -2^0 = -1$

The value 4 has the same zero mantissa (exact binary number) and the exponent 129:

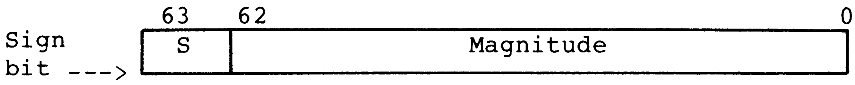
$2^{129-127} = 2^2 = 4$

But 0.25 is always written with a zero mantissa and the exponent 125:

$2^{125-127} = 2^{-2} = 0.25$

This form corresponds to the single precision of the proposed IEEE standard.

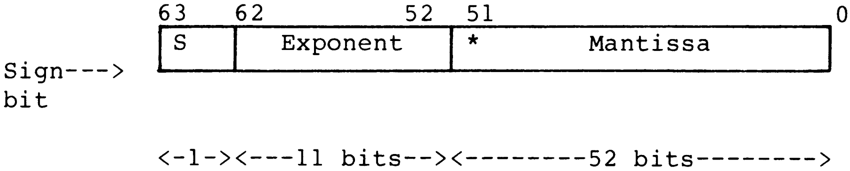
Long integer format



The number is again stored in two´s complement form. The numbers that can be represented range from approximately

-9×10^{18} to $+9 \times 10^{18}$

Long real format

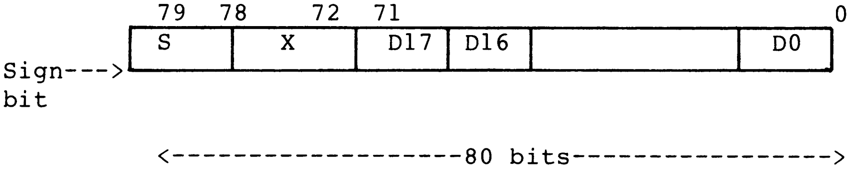

63
62
52
51
0
S
Exponent
*
Mantissa
Sign---> bit
<-1-><---11 bits--><--------52 bits-------->

The exponent is written as an 11-bit number with an offset of 1023 (3FFH).

This form corresponds to double precision (IEEE standard) and allows values over the approximate range of

2.2×10^{-308} to 1.8×10^{308}

BCD format


79
78
72
71
0
S
X
D17
D16
D0
Sign---> bit
<-----------------80 bits----------------->

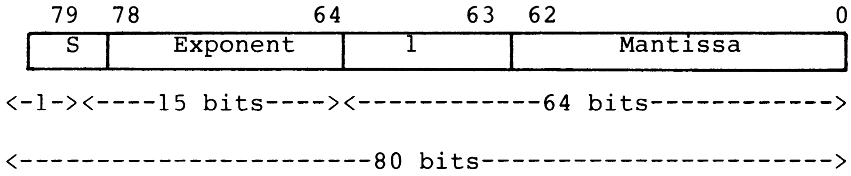
Each of the 18 digits occupies 4 bits and must contain a value between 0 and 9.

In this form 8087 can read or write a packed decimal integer of 18 digits.

Bits 72 to 78 have no significance.

This form is very useful for data input/output but occupies more memory space.

Temporary real format


79
78
64
63
62
0
S
Exponent
1
Mantissa
<-1-><----15 bits----><------------64 bits------------>
<----------------------80 bits---------------------->

The 15-bit exponent has an offset of 16383 (3FFFH).

The mantissa occupies 64 bits but includes the 1 in front of the ´binary point´.

The values that can be represented range from

$$3.4.10^{-4932} \text{ to } 1.2.10^{4932}$$

in absolute value.

This format is only used in memory to store intermediate results when the internal stack is full.

All internal operations are carried out using this format. Other formats are only used for input and output.

Consequently, there is little danger of a rounding error during a calculation, even with double precision real calculations.

Zeros and infinity

By convention, zero is represented by a zero exponent and a zero mantissa. However, the sign retains its 0 or 1 value. There is therefore a +0 and a -0! The value -0 can present formatting problems when a Pascal or Fortran program is involved.

Infinite values can be signed or unsigned, depending on the choice made of the control word. They are represented with an exponent in which all the bits are at 1 and with a zero mantissa (apart from the implicit ´1´).

Two alternative methods of infinity control are supported by the 8087; these may be selected by the setting of the IC field in the control word.

The AFFINE CLOSURE mode will present signed infinite values.

The PROJECTURE CLOSURE mode will present unsigned infinite values; this mode is the most frequently used, and is selected by default.

Special values

NAN (Not A Number)

This signed number has all exponent bits at 1 and any value in the mantissa (apart from 0 which is reserved for the representation of infinity). It is generated by

Stack overflow: writing a number to a full stack
Stack underflow: reading a number from an empty stack
Indeterminate: 0/0, $(-X)^{1/2}$, Log (-X) etc
Use of an operand already in NAN form.

Denormalised number

A denormalised number appears when there is an underflow and the corresponding interrupt is masked.

Underflow occurs when the exponent is forced to 0 (the minumum value for a representation with offset), but in contrast to the zero case, the mantissa remains non zero.

Because all intermediate calculations are carried out using temporary real format, this state of affairs only occurs at the time of input/output and in particular when intermediate results of a program are stored in memory outside the stack in a format other than temporary real.

Unnormal number

This number exists only in temporary real format; it is the ´descendant´ of a denormalisation and results from a masked underflow.

It is held to be valid for subsequent calculations. Depending on the operation, this ´unnormality´ can either disappear or become greater. Thus, the addition of a small unnormal number to a larger normal number produces a normal result. On the other hand, a multiplication of such numbers would produce an unnormal result.

7.2 Microprocessor Architecture

The NPX is divided into two parts, the Control Unit (CU) and the Numerical Execution Unit (NEU). The NEU executes all the numerical instructions, while the CU receives and decodes the instructions, reads and writes the memory operands and executes the control instructions.

Both units work independently, which allows the NPX to maintain synchronisation with the CPU when NEU executes an instruction.

7.2.1 Control unit

The CU synchronises the 8087 operations with the CPU in order to follow the program sequence.

The CPU ensures that all instructions are fetched.

By reading the status lines ($\overline{S2}$-$\overline{S0}$,S6) of the CPU, the CU follows the fetching and decoding operations.

It reads the code at the same time as the CPU and controls an instruction queue identical to that of the CPU. But to be able to do this, the 8087 must know whether it is a 8088 or 8086 CPU. This may be determined by monitoring the state of $\overline{BHE}$-S7 line of the CPU immediately after a processor RESET. Once the 8087 has determined the processor type, it adjusts the number of bytes in its instruction queue to that of the processor.

Lines QS0, QS1 allow the CU to obtain and decode the instructions coming from the queue in synchronisation with the CPU.

After decoding, the 8086 executes all the instructions, except those preceded by an Escape instruction, whereas the 8087 executes only those in this category.

The CPU however looks after the addressing of the ESC instructions. It distinguishes between those that include a memory reference and those that do not.

If the ESC instruction requires a memory operand, the CPU calculates the operand address and carries out a dummy read at this location (it ignores the contents of the word that it reads). All the addressing modes of the 8086/88 are available for the 8087 instructions since it is the CPU which provides all the required addressing.

If the ESC instruction does not contain a memory reference, the CPU simply executes the next instruction.

An ESC instruction either makes no memory reference or requires the reading or writing of one or more memory words. In the first case, the instruction allows the 8087 instruction to be executed immediately.

In the second case, the CU uses the dummy read to latch and save the address provided by the CPU.

If the instruction is a read, the CU also latches the data when it is available on the bus. If the data is longer than a word, the CU immediately obtains the bus using the Request/Grant protocol and reads the remainder of the data, incrementing the address previously latched from the address bus.

When writing, the 8087 ignores the data that follows the dummy read. When it is ready for the write, it obtains control of the bus and writes the operand starting at the address that has been latched.

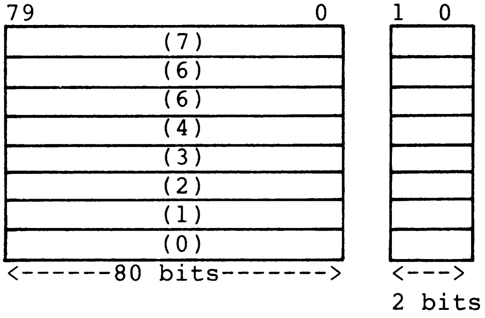
7.2.2 Numeric Execution Unit

The NEU executes all the instructions which modify registers, that is

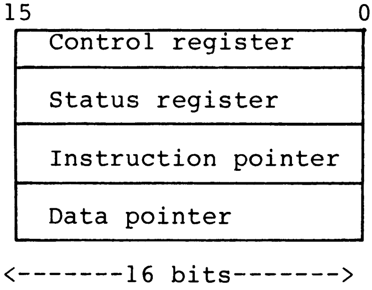
arithmetic instructions
logic (comparison) instructions
transcendental instructions
instructions to transfer constants or data.

7.2.3 Registers

The set of registers accessible to the programmer is shown below

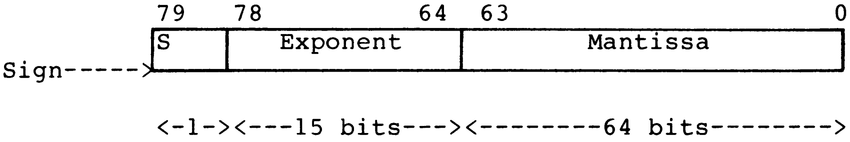

79
0
(7)
(6)
(6)
(4)
(3)
(2)
(1)
(0)
<------80 bits------->
1 0
<--->
2 bits

Register stack Tag word


15
0
Control register
Status register
Instruction pointer
Data pointer
<-------16 bits------->

General Registers

Each of the 8 registers of the 8087 stack has a length of 80 bits; it is divided into a field corresponding to the ´temporary reals´.


79
78
64
63
0
S
Exponent
Mantissa
Sign----->
<-1-><---15 bits---><---------64 bits-------->

7.2.4 Description and use of the stack

At any time, the TOP field (bits 13-11) of the status register identifies the top stack register.

A load operation (push) on the stack decrements TOP by 1 and loads the data into the new top of the stack.

Inversely, a write operation stores in memory the value contained in the top of the stack and increments TOP by 1.

Like the 8086/88 stack, the 8087 stack grows down towards the lower addressed registers.

The instructions may address registers explicitly or implicitly.

Some instructions operate on the register, pointed to by TOP, called ST or ST(0); others allow the programmer to specify one of the registers. An explicit register is always addressed in relation to the top of the stack, with the aid of an index

ST(0) Top of the stack
ST(1)

ST(7)

It is up to the programmer to follow the evolution of indices as a function of the input and output operations of the data. Thus an intermediate result sees the index that points to it change its value frequently.

It is important to emphasise at this point that the most frequently committed error lies in allowing the results of intermediate calculations to overflow the stack. The latter is quickly filled and loading operations will cause denormalised numbers to appear.

It is therefore necessary to control each series of calculations so that the stack is completely emptied after the output of the last result of the series. If necessary, unrequired values can be cleared by an initialisation.

7.2.5 Status word

This register reflects the status of the NPX. Using instruction FSTSW its contents can be stored in memory where the CPU can read it.

The 16-bit status word is divided into the following fields

D15	D14	D13 D11	D10		D8	D7	D6	D5	D4	D3	D2	D1	D0
B	C3	TOP	C2	C1	C0	IR	X	PE	VE	OE	ZE	DE	IE

B (Busy)

Indicates either that the NEU is executing an instruction (B = 1) or that it is waiting (B = 0).

Some instructions are executed exclusively by the CU and do not affect the BUSY bit.

C3,C2-C0

These condition codes are the NPX ´flags´. Their meaning depends on the type of instruction executed, and they are most commonly used to control conditional branching.

Test and comparison

FCOM and similar instructions compare the stack top (ST) register with the operand. The instruction FTST carries out the same operation but with 0 as the operand. This result is described by C3 and C0, with C2 and C1 undefined

C3	C0	
0	0	ST > operand
0	1	ST < operand
1	0	ST = operand
1	1	ST ? operand

Examine

In response to the instruction FXAM, which reports the contents of ST, bits C3, C2, C1 and C0 can take the following values

C3	C2	C1	C0	Result
0	0	0	0	+UNNORMAL
0	0	0	1	+NAN
0	0	1	0	-UNNORMAL
0	0	1	1	-NAN
0	1	0	0	+NORMAL
0	1	0	1	+INFINITY
0	1	1	0	-NORMAL
0	1	1	1	-INFINITY
1	0	0	0	+0
1	0	0	1	EMPTY
1	0	1	0	-0
1	0	1	1	EMPTY
1	1	0	0	+DENORMAL
1	1	0	1	EMPTY
1	1	1	0	-DENORMAL
1	1	1	1	EMPTY

Remainder

The FPREM (Partial REMainder) instruction carries out the ST operation modulo ST(1), and bit C2 above indicates when the operation is complete.

The reduction is in fact ´constrained´ to a factor of 2^{64} in a single operation or can be temporarily suspended to allow another task to be performed.

C2 = 1 the reduction is complete (the remainder is less than ST(1)).

C2 = 0 the reduction is incomplete, so the FPREM instruction must be executed again.

TOP. This 3 bit field shows the absolute number of the register that is currently held as stack top. After initialisation, this number is 7.

IR. Interrupt request bit. It is set to 1 by the NPX to indicate a pending interrupt to the CPU. It can be read by the CPU if the interrupt request passing via the 8259A is inhibited.

D6 is reserved.

D5-D0 indicate the exceptions that may have caused an interrupt. They are assigned as follows.

PE. Precision. The result cannot be represented in the specified format. If the exception is masked, the calculation continues since it is always carried out in temporary real.

UE. Underflow. The result is not zero, but it is too small to be represented in the specified form. If the exception is masked, the 8087 denormalises the number and continues the calculation.

OE. Overflow. The result is too large for the specified form. If the exception is masked, the 8087 treats this number as an infinite number.

ZE. Zerodivide. The divisor of a non-zero finite number is zero. The 8087 treats the result as an infinite number.

DE. Denormalised operand. One of the operands or the result is denormalised. Execution continues if the exception is masked.

IE. Invalid operation. This exception can be generated by

attempts to load to a full stack (stack overflow)
attempts to read from an empty stack (stack underflow)
indetermination (0/0, $(-X)^{1/2}$, Log(-X) etc)
use of an operand that is not a number (NAN)

The result is then a NAN.

7.2.6 Tag words

Tag words mark the contents of each register. Their function is to optimise the operations of the NPX and they allow the contents of a register to be interpreted.

They need not often be used by the programmer.

15			08	07			00
TAG(7)	TAG(6)	TAG(5)	TAG(4)	TAG(3)	TAG(2)	TAG(1)	TAG(0)

Tag values

00 = Valid (normal or ´unnormal´)
01 = Zero
10 = Special (NAN, denormal, infinite)
11 = Empty

7.2.7 Control word

15					08	07							00
X	X	X	IC	RC	PC	M	X	PM	UM	OM	ZM	DM	IM

X Indicates a reserved bit

IC (Infinity Control)
0 = unsigned infinity - PROJECTIVE CLOSURE
1 = signed infinity - AFFINE CLOSURE

RC (Rounding Control)
00 = Rounded to nearest
01 = Rounded down (towards minus infinity)
10 = Rounded up (towards plus infinity)
11 = Truncate (towards zero)

PC (Precision Control)
00 = 24 bits (simple precision)
01 = (reserved)
10 = 53 bits (double precision)
11 = 64 bits (temporary real)

Calculations are not speeded up by choosing a lower precision. It is therefore preferable to choose maximum precision. This precision control is only used to detect format overflow for 8087 inputs and outputs.

M (interrupt Mask)
1 = interrupt towards the 8259A is masked.

D6-D0. Selective masking; the writing of a 1 to the appropriate bit allows each of the exceptions to be selectively masked.

7.3 Instruction Set

The instructions described in this section may be freely mixed with the CPU instructions.

The data addressing mode is the same, only the set of working registers differs.

All the exchanges of data between the CPU and the NPX, including the result of comparison tests, must be made via memory read or write operations.

7.3.1 Data transfer instructions

Format conversions are made implicity at the time of the memory read or write operation.

Real transfers

FLD (source). After decrementing the stack pointer, FLD loads on to top of the stack either a temporary real from a register or from memory, or a single or double precision real from memory. The format is a function of the type of the source operand.

Table 7.1 Data Transfer Instructions

		Possible exceptions						Time (5 MHz)
	Real transfers	I	D	Z	O	U	P	µs
FLD	Load real to ST	X	X					4-15.8
FST	Store real from ST	X			X	X	X	3.6-12.2
FSTP	Store real and POP	X			X	X	X	4-15.8
FXCH	Exchange contents of ST and ST(i)	X						2.4
	Integer transfers	I	D	Z	O	U	P	
FILD	Integer load to ST	X						12.8-16.8
FIST	Integer store from ST	X					X	20-21.2
FISTP	Integer store from ST and POP	X					X	20.2-24.4
	BCD transfer	I	D	Z	O	U	P	
FBLD	Load BCD integer to ST	X						62.2-64.4
FBSTP	Store BCD integer from ST and POP	X						108.4-110.8

I invalid operand
D denormalised operand
Z zerodivision
O overflow
U underflow
P precision error

The instruction FLD ST(0) replicates the value on the top of the stack and allows one to have the same value in both ST(0) and ST(1).

This operation is very useful before output instructions that necessarily include a POP, such as FSTP, FISTP or FBSTP. (See examples 4.4 and 4.15a.)

FST (destination). Copies the value at the top of the stack (ST(0)) into another register or writes its contents in memory, in single or double precision. The format chosen depends on the type of the destination operand (see example 4.13).

FSTP (destination). Carries out the same operation as FST but also pops the stack following the transfer.

Unlike FST this instruction allows a temporary real variable to be stored in memory (see examples 4.4 and 4.15a).

FXCH ST(i) (or nothing). Swaps the contents of the internal register specified with ST(0). If there is no operand, the swap takes place between ST(0) and ST(1). This instruction allows instructions that only operate on the top element of the stack to access other variables.

Integer transfers

FILD (source). After decrementing the stack top, this instruction loads into ST an integer (word, short or long) located in memory, the precise type being dependent on the source operand (see examples 4.6 and 4.15a).

FIST (destination). Copies ST into memory in the word or short integer format, as a function of the type of the destination operand.

FISTP (destination). Carries out the same operation as FIST but also pops the stack. It is only possible to write in the long integer format with this instruction.

Packed decimal transfers

FBLD (source). After decrementing the top of the stack, this instruction loads into ST(0) an integer coded in BCD format from memory.

FBSTP (destination). Writes ST in memory in the form of an integer coded in BCD format. The stack is then popped.

7.3.2 Arithmetic instructions

These instructions include two operands, explicit or implicit; the one on the right is the source, the one on the left is the destination containing the result after the operation.

Of necessity, one of the two is ST(0).

This order can be reversed for subtraction or division.

Table 7.2 Arithmetic Instructions

		Possible exceptions						Time (5 MHz) µs
	Addition	I	D	Z	O	U	P	
FADD	Add real	X	X		X	X	X	17-26
FADDP	Add real and POP	X	X		X	X	X	18
FIADD	Integer add	X	X		X		X	25.4-28.2
	Subtraction	I	D	Z	O	U	P	
FSUB	Subtract real	X	X		X	X	X	17-26
FSUBP	Subtract real and POP	X	X		X	X	X	18
FISUB	Integer subtract (in memory)	X	X		X		X	25.4-28.2
FSUBR	Subtract real reversed a real	X	X		X	X	X	17.4-26
FSUBRP	Subtract real reversed and POP	X	X		X	X	X	18
FISUBR	Integer subtract reversed	X	X		X		X	25.4-28.2
	Multiplication	I	D	Z	O	U	P	
FMUL	Multiply real	X	X		X	X	X	19.4-36.2
FMULP	Multiply real and POP	X	X		X	X	X	20-28.4
FIMUL	Integer multiply	X	X		X		X	27.6-30.4
	Division	I	D	Z	O	U	P	
FDIV	Divide real	X	X	X	X	X	X	39.6-49
FDIVP	Divide real and POP	X	X	X	X	X	X	40.4
FIDIV	Integer divide	X	X	X	X	X	X	47.4-50.4
FDIVR	Divide real reversed	X	X	X	X	X	X	39.8-49.2
FDIVRP	Divide real reversed and POP	X	X	X	X	X	X	40.6
FIDIVR	Integer divide reversed	X	X	X	X	X	X	47,4-50.6

I invalid operand

D denormalised operand

Z zerodivision

O overflow

U underflow

P precision error

The rules for writing are as follows

Form	Operands Destination <> Source	Example
Implicit	{ST(1),ST}	FADD
Register	ST(i),ST or ST,ST(i)	FSUB ST,ST(3)
Register with POP	ST(i),ST	FMULP ST(2),ST
Real in memory	{ST},short or long real	FDIV AZIMUTH
Integer in memory	{ST},word or short integer	FIDIV N_PULSES

In this table the { } symbols indicate the implicit operands and are normally omitted.

Addition

FADD
FADDP
FIADD

The addition instructions, with a real (FADD), with a real with POP (FADDP) or with an integer (FIADD), add the source and the destination and write the sum to the destination.

ST can be doubled by coding the following

```
FADD ST, ST(0)
```

(See examples 4.8 and 4.15a)

Subtraction

FSUB	**FSUBR**
FSUBP	**FSUBRP**
FISUB	**FISUBR**

If the two operands of a subtraction are designated A and B, the result of the operation can be either A - B or B - A. The direction therefore has to be specified as normal or reversed (R). The calculation is separated into two parts.

(1) The operation between ST(0) and the other operand (op.) is carried out in an intermediate register.

For a normal, direct subtraction (FSUB, FSUBP and FISUB), (ST(0) - op.) is calculated. In reversed

subtraction (FSUBR, FSUBRP and FISUBR), (op. - ST(0)) is calculated.
(2) The result of this calculation is copied in the destination.

Thus

```
FSUB  ST(0), op.   ->  ST(0) - op. in ST(0)
FSUBR ST(0), op.   ->  op. - ST(0) in ST(0)
FSUB  op., ST(0)   ->  ST(0) - op. in op.
FSUBR op., ST(0)   ->  op. - ST(0) in op.
```

It is therefore important not to mix the different ways of writing operands with the presence or absence of R. Unfortunately, the manufacturers´ programmer´s reference gives little information on the matter.

Multiplication

FMUL
FMULP
FIMUL

These multiplication instructions, with a real (FMUL), with a real and POP (FMULP) and with an integer (FIMUL), multiply the source and the destination and write the product to the destination.

The following code squares the contents of ST

```
FMUL ST, ST(0)
```

(See examples 4.13 and 4.15a)

Division

FDIV **FDIVR**
FDIVP **FDIVRP**
FIDIV **FIDIVR**

If the two operands of a division are designated A and B, the result of the operation can be either A/B or B/A. Again, the direction has to be specified as normal or reversed (R). The calculation is in two parts.
(1) The operation between ST(0) and the other operand (op.) is carried out in an intermediate register.

For normal direct division (FDIV, FDIVP and FIDIV), (ST(0)/op.) is calculated.
(2) The result of this calculation is copied in the destination.

Thus

```
FDIV  ST(0), op.  ->  ST(0)/op. in ST(0)
FDIVR ST(0), op.  ->  op./ST(0) in ST(0)
FDIV  op., ST(0)  ->  ST(0)/op. in op.
FDIVR op., ST(0)  ->  op./ST(0) in op.
```

As before, it is important not to mix the different ways of writing operands with the presence or absence of R. The manufacturers´ documentation is not very explicit on the subject.

Other functions

FSQRT (without operand). Replaces the contents of ST with its square root.

The square root of -0 is -0 (see example 4.19a).

FSCALE (without operand). Interprets the value contained in ST (1) as an integer lying between -2^{15} and $+2^{15}$ (rounded if necessary) and carries out the following operation

$$ST \leftarrow ST * 2^{ST(1)}$$

This instruction, which only executes one addition on the exponent of ST, allows multiplications with powers of 2 to be carried out very rapidly. It is preferable to load this power of 2 with the aid of a WORD format in order to avoid any error.

FPREM (without operand). Carries out the following operation

ST <- ST MODULO ST(1)

If ST is much larger than ST(1), it may be necessary to divide up this operation in order not to block interrupts or other calculations for too long.

Flag C2 of the status word allows execution of the operation if there is a task change or if the magnitude difference exceeds 2^{64}.

C2 = 1 indicates that the operation is not terminated, and FPREM can be re-executed because ST contains the partial result.

The three bits C3, C1 and C0 give the three least-significant bits of the quotient generated by FPREM.

This instruction is often used together with instructions for trigonometrical operations.

FPTAN, for example, operates only on angles lying between 0 and $\pi/4$. Using $\pi/4$ modulus, ST yields an angle lying between these values and bits C3, C1 and C0 of the status word that indicate the semi quadrant used from the eight possible ones. For example

$\theta = 190^{\circ} = 1.056\pi$
$\pi/4$ is entered in ST(1) and θ in ST.

After execution of FPREM, ST contains 0.056π (10°) = module remainder and C3, C1, C0 = 100(4) = three least-significant bits of the quotient.

FRNDINT rounds ST to a number representing an integer, according to the indications of precision and rounding contained in the control word.

FXTRACT (without operand). Decomposes the number contained in ST into two numbers, carrying out a loading operation (PUSH).

The new ST contains the signed mantissa of the old ST in the form of a real number with zero exponent.

ST(1) contains the signed, true exponent of the initial number, the offset having been suppressed.

If the original number is zero, ST and ST(1) will also be zero but with the same sign as the initial zero number (+0 or -0).

This instruction is very useful for converting real numbers to be output using FBSTP. It is thus possible to obtain successively a representation of the mantissa followed by the exponent.

Table 7.3 Arithmetic Instructions

		Possible exceptions						Time (5 MHz)
	Other functions	I	D	Z	O	U	P	µs
FSQRT	Square root of ST	X	X				X	36.6
FSCALE	Multiply/divide ST by 2^i	X			X	X		7
FPREM	Divide ST modulo ST(1)	X	X			X		25
FRNDINT	Transform ST into an integer (rounded)	X					X	9
EXTRACT	Separate ST into mantissa and exponent	X						10
FABS	Absolute value of ST	X						2.8
FCHS	Change sign of ST	X						3

I invalid operand
D denormalised operand
Z zerodivision
O overflow
U underflow
P precision error

FABS (without operand). Makes the ST sign positive.

FCHS (without operand). Reverses the sign of ST.

7.3.3 Comparison instructions

These instructions modify the C3, C2, C1 and C0 flags of the status word.

It is necessary to transfer the status word into memory with the aid of FSTSW, described later, in order for the CPU to analyse it and make execution decisions.

The write operation must be carried out immediately after the comparison instruction, in order to avoid modification of the flags by other instructions.

Table 7.4 Comparison Instructions

		Possible exceptions						Time (5 MHz)
	Comparison	I	D	Z	O	U	P	µs
FCOM	Compare real	X	X					9-18
FCOMP	Compare real and POP	X	X					9.4-18.4
FCOMPP	Compare real and POP twice	X	X					10
FICOM	Integer compare	X	X					17.4-20.2
FICOMP	As above, but with POP	X	X					17.8-20.6
FTST	TEST	X	X					8.4
FXAM	Examine							3.4

I invalid operand
D denormalised operand
Z zerodivision
O overflow
U underflow
P precision error

FCOM compares ST and the operand, which may be a register or a short or long real in memory.

If there is no operand indicated, ST(1) is used implicitly.

C3 and C0 give the result of the comparison.

FCOMP operates like FCOM but clears ST with a POP.

FCOMPP operates like FCOMP but clears ST and ST(1) by executing two successive POPs.

This instruction does not include any operand because it necessarily works with ST and ST(1).

FICOM converts the word or short integer operand memory to a temporary real and compares it to ST.

FICOMP operates like FICOM but clears ST with a POP.

FTST compares ST to zero and sets flags C3 and C0.

C3	C0	Result
0	0	ST > 0
0	1	ST < 0
1	0	ST = 0
1	1	ST is a NAN or unsigned infinity

FXAM reports the contents of ST and sets flags C3, C2, C1 and C0 of the status word to conform with the table shown above when the status register was described.

7.3.4 Transcendental instructions

These operations provide the basic set necessary for all common trigonometric calculations. Transcendental instructions assume that their operands are both valid and in range. It is necessary for pre- and post-processing software to reduce arguments to the acceptable range and to adjust the result to correspond to the original argument.

These instructions operate on the top one or two elements of the stack only.

FPTAN (without operand). Calculates the tangent of an angle lying between 0 and $\pi/4$, written in radians in ST.

The result is presented in the forms of a ratio Y/X with Y in ST(1) and X in ST, after a loading operation.

This provision allows the calculation of other trigonometric functions to be optimised, using classical trigonometric formulae.

FPATAN (without operand). Calculates the arctangent function of a ratio Y/X. X is taken from ST and Y from ST(1). Y and X must observe the following conditions

$$0 < Y < X$$

with X and Y not being ´infinite´.

After a POP operation, the result is written in radians in ST.

Table 7.5 Transcendental Instructions

	Transcendental function	Possible exceptions I	D	Z	O	U	P	Time (5 MHz) µs
FPTAN	$Y/X=TG(\theta):\theta$ and X in ST Y in ST(1)(PUSH)	X					X	90
FPATAN	$\theta=ARCTG(Y/X):\theta$ and X in ST Y in ST(1)(POP)					X	X	130
F2XM1	$Y=2^X-1$:X and Y in ST					X	X	100
FYL2X	$Z=Y.LOG_2X$:X and Z in ST Y in ST(1)(POP)						X	190
FYL2XP1	$Z=Y.LOG_2(X+1)$:X and Z in ST Y in ST(1)(POP)						X	170

I invalid operand
U underflow
P precision error

F2XM1 (without operand). Calculates the function

$$Y = 2^X - 1$$

where X is taken from ST and must be between 0 and 0.5 inclusive.

Y then replaces X in ST.

This instruction allows a very accurate result to be obtained even when X is close to 0.

To obtain

$$Y = 2^X$$

add 1 (provided by FLD1).

The following functions can also be derived from F2XM1.

$$10^X = 2^{X.LOG_2 10}$$

$$e^X = 2^{X.LOG_2 e}$$

$$Y^X = 2^{X.LOG_2 Y}$$

The constants $LOG_2 10$ and $LOG_2 e$ are provided by the instructions FLDL2T and FLDL2E.

The value of $LOG_2 Y$ can be obtained from FYL2X (see next instruction).

FYL2X (without operand). Calculates the function $Z = Y.LOG_2X$ where X is in ST and Y in ST(1).

X must be positive and not infinite.
Y cannot be less than, nor more than infinity.
Z is written in ST, after a POP operation.
This function optimises the LOG calculation to any base, since a simple multiplication involving a constant is necessary

$$Z = LOG_n(X) = LOG_n2 * LOG_2X$$

FYL2XP1 (without operands). Calculates the function

$$Z = Y * LOG_2(X + 1)$$

X is taken from ST and must observe the following condition

$$0 < | X | < 1 - \frac{\sqrt{2}}{2}$$

Y is taken from ST(1) and must lie between minus infinity and plus infinity.
Z is written in ST, after a POP operation.
This function allows a more accurate result to be obtained, compared with FYLZX, when X is a value very close to 1.

7.3.5 Constants

The constant instructions, which are written in temporary real, are accurate to some 19 decimal places, and are built-in in the 8087. Appreciable savings in execution time can be achieved by using them.
The following instructions load ST

FLDZ	+0.0
FLD1	+1.0
FLDPI	$+\pi$
FLDL2T	LOG_210
FLDL2E	LOG_2E
FLDLG2	$LOG_{10}2$
FLDLN2	LOG_e2

(See example 4.8.)

Table 7.6 Constant Instructions

	Constant	Possible exceptions						Time (5 MHz) µs
		I	D	Z	O	U	P	
FLDZ	+0.0 in ST	X						2.8
FDL1	+1.0 in ST	X						2.8
FLDPI	Pi in ST	X						3.8
FLDL2T	$\mathrm{LOG}_2 10$ in ST	X						3.8
FLDL2E	$\mathrm{LOG}_2 e$ in ST	X						3.6
FLDLG2	$\mathrm{LOG}_{10} 2$ in ST	X						4.2
FLDLN2	$\mathrm{LOG}_e 2$ in ST	X						4

I invalid operand

7.3.6 Processor control instructions

For many of these instructions there are two possible notations. One generates the instruction code, preceded by a WAIT, as do all the preceding instructions implicity. The other generates the instruction code preceded by a NOP instead of the WAIT.

These instructions include an ´N´ in the instruction character string, and are intended for use where the normal form carries an unacceptable risk of precipitating an endless WAIT condition.

In the event of an error, the 8087 generates an interrupt for the CPU and waits for its acknowledgement before continuing. If interrupts are inhibited, the CPU will execute the next instruction and, after decoding the WAIT instruction, may have to wait until the 8087 frees the TEST pin.

The only escape from such a situation is a NMI or a RESET.

Control instructions, such as writing the status word in memory, are often placed after instructions that may cause errors.

If the interrupts are masked, type ´N´ instructions are therefore to be preferred.

Table 7.7 Control Instructions

	Control	Possible exceptions	Time (5 MHz) µs
FINIT or FNINIT	Initialisation		1
FDISI or FNDISI	No interrupt		1
FENI or FNENI	Interrupt allowed		1
FLDCW	Load control word located in memory		3.4-4.8
FSTCW or FNSTCW	Store control word		4.6-6.2
FSTSW or FNSTSW	Store status word		4.6-6.2
FCLEX or FNCLEX	Clear exceptions		1
FSTENV or FNSTENV	Store environment		11.8-14.6
FLDENV	Load environment		10.6-13.2
FSAVE or FNSAVE	Save state		52.8-63.2
FRSTOR	Restore state		52.6-63.6
FINCSTP	Increment stack pointer (POP)		1.8
FDECSTP	Decrement stack pointer (PUSH)		1.8
FFREE	Free register ST(i)		2.2
FNOP	No operation		2.6
FWAIT	CPU wait while BUSY = 1 (=TEST)		1.6(1)

(1) Test for BUSY every 1 µs.

FINIT/FNINIT initialises the NPX in the same way as a RESET. Synchronisation with the CPU is preserved.

After initialisation, the registers are not altered but are considered to be empty (all the TAG bits are set).

All of the bits of the status word are reset to 0, apart from bits C3, C2, C1 and C0 which are indeterminate.

The control word gives the following configuration

unsigned projective infinity
rounded as accurately as possible
64-bit (maximum) precision
masked interrupts
all the exceptions are masked.

At the end of the operation the NPX is initialised (see example 4.13).

FDISI/FNDISI (disable interrupts) inhibits interrupts by setting bit M of the control word to 1.

FENI/FNENI (enable interrupts) enables interrupts by setting M to 0.

FLDCW replaces the control word by the word defined by the operand. Exceptions should be cleared beforehand in order not to risk an immediate interrupt if the new control word has interrupts enabled.

FSTCW/FNSTCW (store control word) copies the status word into the location defined by the operand.

FSTSW/FNSTSW (store status word) writes the current value of the status word into a memory location defined by the destination operand.

This instruction is used in particular

- to make the CPU carry out conditional jumps following a test made by the 8087.
- to test the state of the 8087 (BUSY) without stopping execution of the program, as in the case of the WAIT instruction (FNSTSW).
- to recognise possible exceptions when the interrupts are inhibited.

FCLEX/FNCLEX clears the exceptions, as well as possible interrupt requests and the busy flag of the status word.

This instruction must be sent systematically by the exception handling routines, in order to avoid a permanent loop.

FSAVE/FNSAVE (save state) copies the full state of the 8087, including register contents, into a 94-byte memory location pointed to by the operand.

The following are output, in order

the control word (16 bits)
the status word (16 bits)
the TAG word (16 bits)
the instruction pointer (32 bits) of the current instruction
the data pointer (32 bits) of the current operand
the registers from ST to ST(7) (8 * 10 bytes)

This operation is very useful for saving the full state of the NPX and initialising it for a new ´interrupt´ type procedure.

Some precautions must be taken when using these instructions.

Taking account of the necessary access time, even in DMA (Direct Memory Access) mode, required by the 8087 to load 94 bytes, CPU interrupts need to be disabled.

In addition, it must be ensured that the CPU does not attempt to read or write into a save area before it is completely loaded. The FWAIT (or WAIT) is used for this.

There is no danger of an infinite loop since the NPX exits the FSAVE operation in the initialised state, and therefore without any interrupt request.

FRSTOR (restore state) carries out the reverse of FSAVE. The next following instruction in the NPX must be FWAIT. Therefore, a ´N´ type control instruction must not be used immediately after FRSTOR.

The NPX is exactly in the situation preceding the FSAVE instruction. If necessary, an interrupt can be requested immediately.

FSTENV/FNSTENV (store environment) saves the working environment of the NPX in 14 bytes. This constitutes the contents saved by FSAVE, but without registers ST-ST(7).

There is no initialisation, but all exceptions are masked to avoid infinite loops.

This instruction is normally used by procedures handling exception interrupts of the 8087.

The same precautions as for the FSAVE instruction need to be taken.

FLDENV (load environment) carries out the reverse of the preceding instruction. There must be a FWAIT instruction before the next NPX instruction.

It is also important to ensure that interrupts are masked in the control word before execution of FLDENV.

If necessary, the environment handling routine is made to reset the IEM bit to 0, as well as clear the exception that caused the interrupt.

A re-entry is very probable since a routine that has replied to a 8087 interrupt is usually involved.

FINCSTP (increment stack pointer) adds 1 to the stack top pointer (ST). This corresponds to a POP. The contents of the registers are unaltered.

If ST = 7, then ST = 0 after FINCSTP.

FDECSTP (decrement stack pointer) subtracts 1 from the absolute number on the stack top.

This operation corresponds to a PUSH. The contents of the registers are unaltered.

If ST = 0, then ST = 7 after FDECSTP.

FFREE (free register) frees the register specified in the operand by placing the associated tag in the empty position.

This instruction is very useful to prevent overfilling of the stack with intermediate results.

FNOP No operation.

FWAIT is an alternative for the CPU WAIT instruction. It should be used systematically when the 8087 is loading in memory in order to be certain that the CPU or another bus master cannot access the loading area before the loading is completed.

Appendix

Example A.1

```
!       This program calls for the subroutine assembler KEY so as to execute
!protection with the help of a password. It edits on screen the contents of the
!table of wrong keys filled by the assembler subroutine.

!       In this regard, note how the called program returns the address of a
!table defined by itself to the main call program.


         PROGRAM  PROTECT(INPUT,OUTPUT);
!        ===============================

         TYPE    P=ADS OF ARRAY[1..4,1..6] OF CHAR;

         VAR     KEY_PROTECT:P;
                 I,J:INTEGER;

         FUNCTION KEY(VARS KEY_PROTECT:P):BOOLEAN;EXTERN;

 !

         BEGIN
 !
         IF KEY(KEY_PROTECT) THEN BEGIN
                                          WRITELN;
                                          WRITELN('LETS GO');
                                    END
                               ELSE BEGIN
                                          WRITELN;
                                          WRITELN('REJECTED');
                                    END;
         FOR I:=1 TO 4 DO BEGIN
                                  WRITELN;
                                  FOR J:=1 TO 6 DO WRITE(KEY_PROTECT^[I,J]);
                                  WRITELN;
                           END;
 !
         END.
 !
```

Example A.1a

```
PAGE    62,132
TITLE   KEY
;
;        This procedure extends  example  6.1 by  adding to it the storage of the
;wrong keys in a table. This table is filled with '.' before the trials with  the
;help of the instruction REP STOS. After every unsuccessful trial, an instruction
;REP MOVS transfers the characters making up  the wrong key  from the stack  into
;the table.
;
;        These two repetitive instructions allows counting, index  incrementation
;and data transfers. Moreover, MOVS is the only  instruction that  allows  direct
;memory to memory data transfers (PUSH and POP instructions do the same with just
;one word in the stack).
;
;        Lastly, this procedure is  organized as a  function of a main  PASCAL or
;FORTRAN program. Directly  included in an  IF statement of these  languages,  it
;will return 0FFH for TRUE and 0 for FALSE in the  AL register.  It also  returns
;the address of a table.
;
;        System calls correspond to versions 1.25a or 2.11 of MS_DOS.


                NAME_   KEY
;               ===========

                PUBLIC          KEY

        MS_DOS_CALL     EQU     21H
        DISPLAY_VIDEO   EQU     02H
        KB_WITHOUT_ECHO EQU     08H
        TRIAL_NUMBER    EQU     4
;
        KEY_ST          STRUC
;       --------------------
                        DB      6 DUP(?)        ; Size of a key.
;
        KEY_ST          ENDS
;
        PARAM           STRUC
;       --------------------
                        DB      ?               ; Allows keeping stack alignement
                                                ; (only 8086).
        COUNTER         DB      ?               ; Byte  counting  the  number  of
                                                ; attempts.
        DS_SYSTEM       DW      ?               ; Store  for  the DS  of  calling
                                                ; program.
        KEY_TRIAL       DB      6 DUP(?)        ; Location for putting aside  the
                                                ; trial key.
        FORMER_BP       DW      ?               ; <= BP points this offset.
        RETURN_ADD      DD      ?               ; Return address of this function
        POINTER_ADD     DD      ?               ; Address of pointeur to be  loa_
                                                ; ded with the address of  WRONG_
                                                ; KEY table.
;
        PARAM           ENDS
;
        P               EQU     [BP - OFFSET FORMER_BP]
        OFF_LOCATION    EQU     FORMER_BP - KEY_TRIAL
        PARAM_AREA      EQU     SIZE PARAM - OFFSET POINTER_ADD
;
```

```
  STACK   SEGMENT   STACK 'STACK'
; ===========================
                         DW      128 DUP(?)     ; Additional stack requirement to
                                                ; deal  with  system  calls.  The
                                                ; STACK segment appearing in  the
                                                ; file *.MAP after LINK, is exten_
                                                ; ded by 128 bytes.
  STACK   ENDS

  DIRECTORY   SEGMENT   WORD   PUBLIC   'RAM'
; ======================================
      RIGHT_KEY        DB       'XYZ873'
      END_KEY          LABEL    BYTE
      MESSAGE          DB       0AH,0DH,'PASSWORD OK'
      MESSAGE_END      LABEL    BYTE
;
  DIRECTORY   ENDS

  DATA_TABLE   SEGMENT   WORD   PUBLIC   'RAM'
; ========================================
    WRONG_KEY  KEY_ST  TRIAL_NUMBER DUP(<>)     ; Storage  for  wrong   keys.
                                                ; TYPE = size of structure (6).
                                                ; LENGTH = number of DUP (4 here)
                                                ; SIZE = LENGTH * TYPE (24 here).

  DATA_TABLE   ENDS

  CODE_PP   SEGMENT   BYTE   PUBLIC   'ROM'
; =====================================
        ASSUME  CS:CODE_PP,ES:DATA_TABLE


;
  KEY          PROC    FAR
; ----------------------


               PUSH     BP                      ; Actual content of BP is written
                                                ; in FORMER_BP of PARAM structure
               MOV      BP,SP                   ; BP points the structured stack.
               SUB      SP,OFFSET FORMER_BP     ; Reservation  of  locations  for
                                                ; local variables, in the stack.

               MOV      P.DS_SYSTEM,DS          ; Storage of DS of calling program

               MOV      AX,DATA_TABLE           ; In line with ASSUME.
               MOV      ES,AX

;.......Clearing of the area allocated to wrong keys (used of STOS) :...........

               LEA      DI,WRONG_KEY            ; DI points this area (compulsory)
               MOV      AL,'.'                  ; AL = '.' is the value used  for
                                                ; fill up.
               MOV      CX,SIZE WRONG_KEY       ; CX = total number of bytes.
               CLD                              ; (CLear Direction flag) => DI is
                                                ; increased at every repetition.
         REP   STOSB                            ; WRONG_KEY  cannot  be  written
                                                ; in this instruction,  its  type
                                                ; being that of KEY_ST ( type=6 )
                                                ; and not that of a byte (1). This
                                                ; transfer can be interrupted by a
                                                ; machine interrupt.
```

```
;.......Request for password :..........................................

                MOV     P.COUNTER,0             ; Initialisation of counter.

   TRIAL:       CMP     P.COUNTER,TRIAL_NUMBER  ; Test of the number of trials.
                JNE     OTHER_TRIAL             ; Conditional short jump, limited
                JMP     REJECT                  ; to 127, which imposes  recourse
                                                ; to a  JMP  in  order  to  reach
                                                ; REJECT.
   OTHER_TRIAL: MOV     AH,DISPLAY_VIDEO        ; Screen  display  of  a  message
                MOV     DL,'?'                  ; request ('?').
                INT     MS_DOS_CALL             ; Type  21H  internal   interrupt
                                                ; executed by the system.
                                                ; AH contains the call number and
                                                ; DL the  character to be  trans_
                                                ; mitted.

;.......Reading, without echo, of the characters of password on trial :..........

                MOV     CX,LENGTH KEY_TRIAL     ; LENGTH gives the number of DUP.
                MOV     AH,KB_WITHOUT_ECHO
                XOR     DI,DI                   ; Index initialisation.
   RECEPTION:   INT     MS_DOS_CALL             ; Read character returns to AL.
                MOV     P.KEY_TRIAL[DI],AL      ; AL=>SS:[BP]+offset KEY_TRIAL+DI
                INC     DI
                LOOP    RECEPTION               ; Request as long as CX ≠ 0.

;.......Using CMPS for comparing KEY_TRIAL with RIGHT_KEY :......................
        ASSUME  ES:DIRECTORY

                MOV     AX,DIRECTORY            ; In line with ASSUME.
                MOV     ES,AX


                MOV     DI,OFFSET RIGHT_KEY     ; ES:[DI]  points the  right ope_
                                                ; rand of CMPS. ES is compulsory.
                MOV     SI,BP                   ; SI points the left operand.
                SUB     SI,OFF_LOCATION         ; Its segment depends on ASSUME.
                MOV     AX,SS                   ; No  named  variable:  anonymous
                MOV     DS,AX                   ; CMPBS must be used.  DS is then
                                                ; compulsory.
                MOV     CX,LENGTH KEY_TRIAL     ; LENGTH gives the number of DUP.
                CLD                             ; (CLear Direction flag)=> DI and
                                                ; SI are increased by 1 (CMPSB)
        REPE    CMPSB                           ; Repetition as long as  CX≠0 and
                                                ; bytes of strings are the  same.
                                                ; This transfer may be interrupted
                                                ; without  error if  REPE  is the
                                                ; only prefix.
                JCXZ    CORRECT                 ; Jump if CX = 0.


;.......Storing of the wrong key with instruction MOVS followed by new trial :...
        ASSUME  ES:DATA_TABLE

                MOV     AX,DATA_TABLE           ; In line with ASSUME.
                MOV     ES,AX

   WRONG:       LEA     DI,WRONG_KEY            ; Loading of destination  address
                MOV     AL,TYPE KEY_ST          ; associated with ES (compulsory)
                MUL     P.COUNTER               ; Computing of the  shift in  the
                                                ; table.  Implied  multiplication
                                                ; of AL by a memory byte. The re_
                                                ; sult is written in AX.
                ADD     DI,AX
                MOV     SI,BP                   ; Same condition as for preceding
                SUB     SI,OFF_LOCATION         ; CMPSB.
                MOV     CX,LENGTH KEY_TRIAL     ; LENGTH gives the number of DUP.

        REP     MOVSB                           ; Same remarks as for CMPSB.

                INC     P.COUNTER               ; Increase of counter.
                JMP     TRIAL                   ; New trial.

;.......Sending of message 'PASSWORD OK' :.......................................
        ASSUME  DS:DIRECTORY

  CORRECT:      MOV     AX,DIRECTORY
                MOV     DS,AX

                MOV     CX,MESSAGE_END-MESSAGE  ; Offset   subtraction    giving
                                                ; length of message in bytes.
                MOV     AH,DISPLAY_VIDEO
                XOR     SI,SI                   ; Index initialisation.

  SEND:         MOV     DL,MESSAGE[SI]          ; Sending of character indexed by
                                                ; SI.
                INT     MS_DOS_CALL
                INC     SI
                LOOP    SEND                    ; Return to SEND: as long as CX≠0

;.......Return of a flag into AL: 0 for wrong key and OFFH for right key :.......

                MOV     AL,OFFH
                JMP     COPY_ADD
  REJECT:       XOR     AL,AL

;.......Sending of address of the table WRONG_KEY and return :..................

  COPY_ADD:     LDS     BX,P.POINTER_ADD                ; Loading of the address
                                                        ; of the  pointer  to  be
                                                        ; loaded.
                MOV     WORD PTR [BX],OFFSET WRONG_KEY  ; WORD PTR  is  necessary
                MOV     WORD PTR [BX+2],DATA_TABLE      ; as the  other  operands
                                                        ; are  of  the  immediate
                                                        ; type.
                MOV     DS,P.DS_SYSTEM
                MOV     SP,BP
                POP     BP
                RET     PARAM_AREA
;
 KEY            ENDP
;
 CODE_PP      ENDS
;
                END
```

Index

AAA 147
AAD 150
AAM 149
AAS 149
ADC 147
ADD 147, examples 4.2, 4.21a
AF 112
AGAIN 22
AND 150, example 3.8
ASSUME 33, 44, 70, 73, examples 2.8, 4.2, 4.4, 4.10, 4.12
AT 17
AX 2

BIU 109
BP 2, 23, 78
BUS 3
BX (BH,BL) 2, 23
BYTE 16

CALL 24, 56, 155 examples 2.16, 4.2, 4.19a
CBW 150
CF 112
CLASS 13, 18
CLC 158
CLD 158, 160 example 6.1
CLI 158
CLK 103, 120
CMC 158
CMP (CMPS, CMPSB) examples 4.8, 4.19a, 6.1
COMMON 18
CS 2, 23, 24
CU 154
CWD 150
CX (CH, CL) 2

DAA 147
DAS 149
DB 39
DD 39
DE 174
DEC 147, example 4.8
DENORMALISED 168
DF 113
DI 2, 23, 154
DIV 149
DQ 40
DS 2, 23, 26
DT 41, 107
DUP 25, 44, examples 2.10, 3.6
DW 39
DWORD 165
DX (DH, DL) 2

END 35
ENDP 29
ENDS 16, 19, examples 2.1, 2.2 2.3, 3.10
EQ 67
EQU 37, 47, 68, examples 3.2, 3.8, 4.9, 6.1
ESC 159
ES 2, 23
EU 109
EVEN 20, example 2.5
EXTRN 34, 76, examples 2.17, 4.12

FABS 183
FADD 179
FADDP 179, examples 4.8, 4.15a
FAR 24, example 4.10
FBLD 177
FBSTP 177
FCHS 183
FCLEX 189
FCOM 183
FCOMP 183
FCOMPP 183
FDECSTP 190
FDISI 188

FDIV, FDIVP, FDIVR,
FDIVRP 180
FENI 180
FFREE 190
FIADD 179
FICOM 183
FICOMP 184
FIDIV, FIDIVR 180
FILD 177, examples 4.6, 4.15a
FIMUL 180
FINCSTP 190
FINIT 188, examples 4.13, 4.21a
FIST, FISTP 177
FISUB, FISUBR 179
FLAG 111
FLD 176, examples 4.4, 4.15a, 4.21a
FLDCW 189
FLDENV 190
FLDLG2, FLDLN2, FLDL2E, FLDL2T, FLDPI 186
FLDZ 186, examples 4.8, 4.21a
FLD1 186
FMUL 180, examples 4.13, 4.15a
FMULP 180, example 4.13
FNCLEX 189
FNDISI 189
FNENI 189
FNINIT 188
FNOP 190
FNSAVE 189
FNSTCW 189
FNSTSW 189
FNSTENV 190
FPATAN 184
FPREM 181
FPTAN 182
FRNDINT 182
FRSTOR 190
FSAVE 189
FSCALE 181
FSQRT 181, examples 4.15a, 4.21a
FST 177, example 4.13
FSTCW 189
FSTENV 190
FSTP 177, examples 4.4, 5.15a, 4.21a
FSTSW 189
FSUB, FSUBP, FSUBR, FSUBRP 179
FTST 184
FWAIT 190
FXAM 184
FXCH 177
FXTRACT 182
FYL2X 186
FYL2XP1 186
F2XM1 185

GE 67
GROUP 13, 30, 73, examples 2.16, 4.11
GT 67

HIGH 64, example 4.7
HLT 156

IC 175
ICW 132, 133
IDIV 149
IE 174
IF 113
IMUL 149
IN 145
INC 147, example 4.9
INT 156, examples 4.19a, 5.1
INTO 156
INTR 103
IP 2, 23, 24
IRET 30, 82, 140, 156, examples 2.15, 5.1a

JMP 24, 56, 155, examples 2.6, 4.2
JA, JAE, JB, JBE, JC, JCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ 156, 157, examples 4.2, 4.8, 4.20a, 5.1, 6.1

LABEL 22, example 2.9
LAHF 146
LDS 42, 82, 146, examples 3.4, 4.15a
LE 67
LEA 146, examples 4.8, 4.12, 6.1
LENGTH 65, examples 4.1, 6.1, A1.1
LES 146, example 4.4
LOCK 106
LODS 154

LOOP, LOOPE, LOOPNE, LOOPZ 156, examples 2.8, 3.9, 4.2, 4.8, 4.20a
LOW 64, example 4.7
LT 67

MASK 47, example 3.9
MEMORY 18
MOD 67
MOV, MOVS, MOVSB, MOVSW 145, 154, examples 2.10, A1.1
MUL 149, examples 4.8, A1.1a

NAME 33, example 2.17
NAN 168, 174
NDP 163
NE 67
NEAR 24
NEU 170
NEG 149
NMI 103
NOP 158
NOT 150
NOTHING 73
NPX 163

OCW 134
OE 174
OF 112
OFFSET 65
OR 151, example 4.9
ORG 19, example 2.4
OUT 145

PAGE 17
PARA 16
PC 175
PE 174
PF 113
POINTER 42
POP 145, examples 4.13, 4.15a
POPF 146
PREFETCH 109
PROC/ENDP 29, example 2.14
PTR 55, examples 4.1, 4.6
PUBLIC 17, 34, examples 2.17, 4.12
PUSH 145, examples 4.13, 4.15a

QWORD

RC 175
RCL 150 example 4.21a
RCR 150 example 4.21a
READY 103, 120
RECORD 45, 47, example 3.7
REGISTER 110, 171
REP, REPE, REPNE, REPNZ, REPZ 152, examples 6.1, A1.1
RESET 24, 103
RET 155, examples 2.14, 2.15, 4.15a
ROL 150
'ROM' 18
ROR 150

SAHF 146
SAL 150
SAR 150
SBB 147
SCAS, SCASB, SCASW 154
SEG 64, example 2.17
SEGMENT/ENDS 16, examples 2.1, 2.2, 2.3
SEGMENT PREFIX 63
SF 112
SHL 150, example 4.21a
SHORT 64, example 4.8
SHR 150, examples 3.9, 4.19a
SI 2, 23
SIZE 65, examples 4.8, A1.1
SP 2, 23, 25, 26
SS 2, 23, 25
ST, ST(0), ST(1), ST(2),..., ST(7) 172
STACK 18, 171
STC 158
STD 159
STI 159
STOS, STOSB, STOSW 154, example A1.1
STRUC/ENDS 48, 49, example 3.10
SUB 147, examples 4.13, 4.15a, 4.21a

TAG 175
TBYTE 167
TEST 104, 150
TF 113
THIS 64
TOP 171, 174
TRAP 127

TYPE 36, 49, 65,
examples 4.8, A1.1

UE 174
UNNORMAL 169

WAIT 99,159, example
4.21a

WIDTH 47, example 3.9
WORD 17

XOR 151

ZE 174
ZF 113